Jordi Salvador Pont

Programació en UNIX

per a pràctiques de Sistemes Operatius

Curs 2014–2015

Enginyeria i Arquitectura La Salle

23 de setembre de 2014

Índex

Introducció

Pròleg

> Also be aware that this page is large and is meant to provide a lot of important information. It is not for the *tl;dr;* (too long, didn't read) set of people with minimal attention span. It contains lots of that scary thing called "text" and "information". It is assumed that you can make use of the education you have been provided with that allows you to read and comprehend what has been written.

Introducció

Les pràctiques de Sistemes Operatius tenen una dificultat i una complexitat força elevades, i acostumen a requerir un esforç i una quantitat de temps considerable. Al llarg de la seva realització, l'alumne ha d'aprendre una bona colla d'eines (no només funcions) tant de C com, sobretot de programació en UNIX. El llibre que el lector té a les mans es proposa explicar i ensenyar aquestes eines, de la manera més didàctica possible. Per tant, és altament recomanable que l'alumne prengui el temps necessari per anar llegint cadascun dels capítols a mesura que va realitzant les seves pràctiques. Assumim, però, que disposa d'uns coneixements suficients sobre C, i si no fos així, seria molt aconsellable que refresqués el que ha après als cursos i assignatures anteriors de la carrera.

El material està organitzat en tres parts. A la primera, exposem els continguts més freqüents i que, d'alguna manera, són més senzills d'assolir. Es pot donar pràcticament per fet que el lector llegirà tots aquests capítols, simplement per les necessitats amb què es trobarà mentre faci la pràctica. La segona part és una ampliació i un aprofundiment del material anteriorment exposat, explicant també tècniques alternatives a les de la primera part. La tercera part està dedicada als threads, una novetat recent al contingut de l'assignatura.

Esperem, doncs, que el fruit del nostre esforç serà útil a l'alumne en la realització de les pràctiques de l'assignatura, i restem oberts als comentaris i suggeriments que tinguin a bé fer sobre aquest llibre.

PART I

Eines fonamentals

Capítol 1

Processos en UNIX: introducció

1.1 Introducció

El primer capítol d'aquesta obra versa sobre els *processos*, concepte fonamental en l'ús, disseny i implementació d'un sistema operatiu, i també bàsic per a entendre la resta del material d'aquest llibre. En aquest primer capítol no es pretén explicar tots els detalls relacionats amb un procés, sinó tan sols els més importants i els bàsics per a entendre la resta de capítols d'aquesta primera part dedicada a les eines fonamentals.

La segona part, dedicada als conceptes avançats, també començarà amb un capítol dedicat als processos (capítol 9), on es completarà i s'ampliarà el material presentat en aquest capítol.

1.1.1 Conceptes

Un procés és un programa en execució. Inclou no només les instruccions del programa, sinó també les dades — és a dir, el valor de les variables, el contingut de la memòria, les funcions que s'estan executant, o sigui: el context del programa mentre s'executa. Des del punt de vista del sistema operatiu, un procés consisteix en un espai de memòria que conté el codi del programa, les variables, la memòria ubicada, etc; i un conjunt d'estructures (fora de l'espai de memòria directament accessible pel procés) que guarden informació sobre l'estat del procés, com per exemple la taula de descriptors d'arxiu oberts, senyals pendents, etc.

El lector interessat pot consultar [Kerrisk, 2010, apartats 6.3, 6.4 i 6.5; pàgines 115-122] per a més detalls sobre l'organització de la memòria d'un procés.

1.1.2 Identificador de procés

Cada procés té un identificador de procés, anomenat PID. El PID acostuma a ser un enter. La funció `getpid` retorna l'identificador del procés que crida la funció:

```
#include <unistd.h>

pid_t getpid (void);
```

on el tipus `pid_t` acostuma a ser `int` o similar. Un procés també pot saber l'identificador del seu *pare*, és a dir, del procés que l'ha creat, amb la funció següent:

```
#include <unistd.h>

pid_t getppid (void);
```

Vegi's l'apartat 1.2 per a més detalls sobre el concepte de *procés pare*.

1.1.3 UID i GID

A més de l'identificador de procés, hi ha dos parells d'identificadors que convé tenir en compte:

- UID i GID real.
- UID i GID efectiu.

on UID significa *user identifier* i GID significa *group identifier*. El sistema operatiu UNIX és multiusuari, i organitza els usuaris en grups. Els identificadors UID i GID reals identifiquen a quin usuari i grup pertany el procés (que no el programa!, que pot pertànyer a un altre usuari). A l'hora de determinar quines accions i quins permisos s'atorguen a un procés es tenen en compte l'UID i GID efectiu, que per defecte coincideixen amb l'UID i GID reals — però hi ha circumstàncies en què no és així.

Per a una discussió completa d'aquests identificadors i els seus efectes, el lector pot consultar [Kerrisk, 2010, capítol 9], i el capítol 39 d'aquesta mateixa obra exposa un mecanisme exclusiu de Linux (anomenat *capabilities*) per a portar un control més fi dels permisos atorgats a un procés.

1.2 La funció *fork*

En UNIX, crear un nou procés significa *duplicar-lo*, *clonar-lo*. A partir d'aquí es distingeix entre el *procés pare* i el *procés fill*:

- El procés que crea un nou procés és el procés pare del nou procés creat.

- El nou procés creat és el fill del procés que l'ha creat.

Per a crear un nou procés es disposa de la funció `fork`:

```
#include <unistd.h>

pid_t fork (void);
```

Aquesta funció retorna:

- -1 en cas d'error. Llavors no es crea cap procés nou.
- 0 per al procés fill. Pot consultar el seu propi PID amb la funció `getpid`, i el del seu pare amb `getppid`, com s'ha explicat a l'apartat 1.1.2.
- Per al procés pare, retorna l'identificador del procés fill.

Quan un procés crida la funció `fork`, hi ha un sol procés; un cop retorna, hi ha dos processos idèntics, i el valor de retorn serveix perquè sàpiguen si són el pare o el fill:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int
main (void)
{
  pid_t pid;
  switch ((pid = fork ()))
    {
    case -1:
      perror ("fork");
      exit (-1);

    case 0:
      printf ("I'm the child\n");
      printf ("PID: %d, PPID: %d\n", getpid (), getppid ());
      _exit (0);

    default:
      wait (NULL);
      printf ("I'm the parent\n");
      printf ("Child's PID: %d\n", pid);
      exit (0);
    }
}
```

Encara que hi hagi un sol codi font, hi ha en realitat dos processos. A partir d'aquest punt, és a dir, des del moment en què el pare crea el fill, tant l'un com l'altre evolucionen de manera independent, sense que hi hagi cap zona de memòria que comparteixin (excepte, en tot cas, la zona de memòria d'instruccions i aquelles zones de memòria que explícitament s'hagin creat perquè siguin compartides).

També és important remarcar que el programador no pot saber ni determinar quin dos dels processos (el pare o el fill) serà el que començarà a executar-se. Pot ser el pare, pot ser el fill, o fins i tot poden ser els dos alhora, en sistemes processador multi-core o sistemes multi-processador. Per tant, el programador no pot fer cap assumpció ni fer que el comportament del programa depengui de quin dels dos processos (pare o fill) s'executa primer. Per a una discussió detallada sobre aquest aspecte, es recomana consultar [Kerrisk, 2010, apartat 2.4., pàgines 525–527].

A l'exemple presentat es pot observar que el procés fill finalitza la seva execució cridant `_exit`, mentre que el pare crida `exit`. Quina diferència hi ha entre aquestes dues funcions? Encara és d'hora per a explicar-ho, però sí cal saber que tenen matisos diferents (vegi's l'apartat 9.3). També es pot observar la funció `wait`, que s'explicarà més endavant, a l'apartat 1.4. De moment n'hi ha prou amb saber que el pare crida `wait` per esperar-se que el fill hagi finalitzat la seva execució.

Quan es crea el fill, tots els file descriptors del pare queden duplicats al fill, com si s'hagués cridat la funció `dup`, de manera que els descriptors del pare i del fill comparteixen el *file offset*. És a dir, que si (per exemple) el pare fa una lectura, la següent lectura que faci el fill es farà a continuació de la lectura feta anteriorment pel pare.

1.3 La funció *exit*

L'execució d'un procés pot finalitzar per diverses causes, com per exemple que la funció `main` s'acaba o retorna, o bé es rep un senyal que causa la mort del procés. Hi ha una funció pensada explícitament per a finalitzar un procés:

```
#include <stdlib.h>

void exit (int status);
```

Aquesta funció no és pròpia del sistema operatiu (que ofereix la funció `_exit`, de la qual se'n parla a l'apartat 9.3), sinó de la llibreria de C. La funció `exit` fa els següents passos:

1. Crida els *exit handlers*, que són funcions que s'han registrat perquè es cridin automàticament en aquest moment (vegi's [Kerrisk, 2010, apartat 25.3, pàgines 533-537]).
2. Tanca tots els `FILE *`.
3. Crida la funció `_exit`, passant-li l'argument `status` que ha rebut.

Un procés també finalitza la seva execució quan retorna de la funció `main`, que gairebé sempre és equivalent a cridar la funció `exit` (vegi's [Kerrisk, 2010, pàgina 532] per a més detalls). Els detalls específics sobre quines accions duu a terme el sistema operatiu quan es crida la funció `_exit` es poden consultar a [Kerrisk, 2010, apartat 25.2, pàgina 533].

Com a norma general, un procés ha de finalitzar la seva execució cridant `exit`, però hi ha casos en què és més segur cridar `_exit` (com, per exemple, en el cas de la funció `vfork`). Vegi's l'apartat 9.3 per a més detalls.

1.4 La funció *wait* i família

La funció `wait` (i la seva "família") serveixen per a monitoritzar (que no depurar) l'execució dels fills; la seva principal finalitat és saber si un fill ha finalitzat la seva execució o aturar l'execució fins que un fill finalitzi. En aquest apartat es presentaran les funcions més comunes; a l'apartat 9.5 es presentaran funcions més avançades.

1.4.1 La funció *wait*

La funció `wait` és la més senzilla:

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

pid_t wait (int *status);
```

Al cridar aquesta funció es fa el següent:

1. Si en el moment de cridar la funció no hi ha cap procés fill que hagi finalitzat la seva execució (des de l'últim cop que es va cridar `wait` o alguna funció de la seva "família"), el procés es bloqueja fins que algun fill finalitza.

2. Si en el moment de cridar la funció hi ha un fill que ha finalitzat la seva execució (novament, des de l'últim cop que es va cridar `wait` o alguna funció de la seva "família"), la funció `wait` retorna immediatament el PID del fill finalitzat.

3. Si `status` no és `NULL`, llavors l'enter que apunta contindrà informació sobre les circumstàncies de la mort del fill. Se'n parlarà més endavant en aquest mateix apartat.

En cas d'error, `wait` retorna -1. Un possible error és que no hi hagi cap més fill viu. A l'exemple de la funció `fork` s'ha vist un primer exemple de com cridar la funció `wait`.

1.4.2 La funció *waitpid*

La funció `wait` presenta una sèrie de limitacions:

1. Si un procés té diversos fills, `wait` no permet esperar la mort d'un d'ells en concret, sinó que s'esperarà que algun es mori, sigui quin sigui.
2. Si cap procés fill s'ha mort, `wait` bloqueja el procés. Hi ha circumstàncies en què això pot no interessar-nos.
3. La funció `wait` no permet informar-nos si un fill ha sigut aturat o ha continuat la seva execució (senyals `SIGSTOP` i `SIGCONT`, respectivament).

La funció `waitpid` permet resoldre aquestes limitacions, i està declarada de la següent manera:

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

pid_t waitpid (pid_t pid, int *status, int options);
```

Pel que fa al valor de retorn i a l'argument `status`, `waitpid` funciona igual que `wait`. Les diferències estan en el primer i tercer arguments:

- Si `pid` és més gran que zero, llavors `waitpid` s'ha d'esperar pel fill amb PID `pid`.
- Si `pid` és zero, llavors `waitpid` s'espera per qualsevol fill que estigui al mateix grup de processos que el pare.
- Si `pid` és -1, llavors `waitpid` s'espera per qualsevol fill. De fet, cridar

  ```
  wait (&status);
  ```

  és com cridar

  ```
  waitpid (-1, &status, 0);
  ```

- Si `pid` és un número negatiu diferent de -1, llavors `waitpid` s'espera per qualsevol fill que tingui, com a identificador de grup de processos, el valor absolut de `pid`.

Si no hi ha cap fill que faci "match" amb el que especifica l'argument `pid`, llavors `waitpid` retorna -1 amb el codi d'error `ECHILD` (el codi d'error es posa a la variable global `errno`, declarada a `errno.h`).

Pel que fa les opcions, hi ha les següents disponibles, que es poden combinar amb l'operador `|`:

`WUNTRACED` A més d'informar dels fills morts, informa dels fills aturats per un senyal.

`WCONTINUED` A més d'informar dels fills morts, informa dels fills continuats per `SIGCONT`.

`WNOHANG` Si no hi ha cap fill mort (o aturat o continuat, si és el cas), en lloc de bloquejar el procés, `waitpid` retorna zero.

1.4.3 El paràmetre *status*

Les dues funcions vistes tenen un paràmetre `status`, punter a enter, que permeten que el pare tingui informació sobre les circumstàncies de la mort del fill. Podem distingir entre les següents circumstàncies:

- El fill ha mort cridant la funció `_exit` (o `exit`).
- El fill ha mort perquè ha rebut un senyal que provoca la mort del procés.
- El fill ha sigut aturat per un senyal, i el pare ha cridat `waitpid` amb el flag `WUNTRACED`.
- El fill ha prosseguit la seva execució al rebre `SIGCONT`, i el pare ha cridat `waitpid` amb el flag `WCONTINUED`.

Hi ha una sèrie de macros per a saber quin dels casos és el que s'ha donat en la mort d'un fill. Cadascuna d'aquestes macros rep, com a argument, l'enter apuntat per `status`:

`WIFEXITED` Retorna cert si el fill ha mort cridant `_exit` (o `exit`). Retornar des de la funció `main` és equivalent. Quan aquesta macro retorna cert, es pot emprar la macro `WEXITSTATUS` per saber el codi de retorn, que és l'argument que reben `_exit` i `exit`, o el valor de retorn de la funció `main`.

`WIFSIGNALED` Retorna cert si el fill ha mort per la recepció d'un senyal. En aquest cas, la macro `WTERMSIG` serveix per saber de quin senyal es tracta, i la macro `WCOREDUMP` retorna cert si el fill ha creat un fitxer *core*.

`WIFSTOPPED` Retorna cert si el fill ha sigut aturat per un senyal (generalment `SIGSTOP`). Llavors, la macro `WSTOPSIG` retorna el senyal que ha causat l'aturada del procés.

`WIFCONTINUED` Retorna cert si el fil ha reprès la seva execució a l'haver rebut `SIGCONT`.

1.4.4 Exemple

Com a exemple es presenta una adaptació del codi presentat a [Kerrisk, 2010, pàgines 546–549]. Aquest exemple combina tant l'ús de la funció `waitpid` com el de les macros acabades d'explicar. Aquest exemple consisteix en un programa que pot rebre un argument per línia de comandes, que consisteix en un codi de retorn. El pare crea un fill, i s'espera que mori. Si hi ha codi de retorn, el fill finalitza immediatament amb aquest codi de retorn. Si no, s'espera que arribi un senyal. El pare monitoritza el fill, i quan aquest mor, el pare també mor.

```
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

void
print_wait_status (const char *msg, int status)
{
  if (msg)
    printf ("%s\n", msg);

  if (WIFEXITED (status))
    printf ("child exited, status = %d", WEXITSTATUS (status));
  else if (WIFSIGNALED (status))
    {
      printf ("child killed by %s (%d)", strsignal (WTERMSIG (status)),
              WTERMSIG (status));
      if (WCOREDUMP (status))
        printf (", core dumped");
    }
  else if (WIFSTOPPED (status))
    printf ("child stopped by %s (%d)", strsignal (WSTOPSIG (status)),
            WSTOPSIG (status));
  else if (WIFCONTINUED (status))
    printf ("child continued");
  else
    printf ("no idea");
  printf ("\n");
}

int
main (int argc, char *argv[])
{
  pid_t pid = fork ();
```



```
40    switch (pid)
41      {
42      case -1:
43        printf ("fork: %s (%d)\n", strerror (errno), errno);
44        exit (-1);
45
46      case 0:
47        printf ("Child's PID: %d\n", getpid ());
48        if (argc > 1)
49          exit (atoi (argv[0]));
50        while (1)
51          pause ();
52        exit (0);
53
54      default:
55        while (1)
56          {
57            int status;
58            pid_t child_pid = waitpid (-1, &status, WUNTRACED | WCONTINUED);
59            if (child_pid < 0)
60              {
61                printf ("waitpid: %s (%d)\n", strerror (errno), errno);
62                exit (-1);
63              }
64
65            print_wait_status (NULL, status);
66            if (WIFEXITED (status) || WIFSIGNALED (status))
67              exit (0);
68          }
69      }
70
71    return 0;
72  }
```

1.4.5 Processos orfes i processos "zombie"

El temps de vida d'un procés i dels seus fills acostuma a no coincidir, la qual cosa planteja dues qüestions:

- Quan el pare d'un procés es mor, qui passa a ser el seu nou pare?

  Quan un procés queda orfe, és a dir, el seu procés pare ha mort, el seu nou pare passa a ser el procés `init` (el "predecessor" de tots els processos), el PID del qual és 1.

- Què passa quan un fill es mor?

  Quan un fill es mor es queda en un estat de "zombie". En aquest estat, la immensa majoria dels recursos usats pel procés s'han tornat al sistema operatiu,

però es guarda informació suficient perquè el pare, a l'executar `wait` (o similars), pugui obtenir informació d'aquest fill. Llavors deixa d'estar en zombie i desapareix definitivament.

Què passaria si el pare mor abans que pugui executar `wait`? El fill no es quedarà zombie eternament, sinó que serà adoptat pel procés `init`, el qual cridarà `wait` per fer desaparèixer el zombie.

Per a un procés que crea fills és important cridar `wait`, ja que els processos zombie ocupen una entrada a la taula de processos del sistema operatiu, i podrien arribar a impedir la creació de nous fills. Cal recordar que els zombies no es poden "matar" amb `SIGKILL` ni amb cap altre senyal.

1.5 Execució de nous programes

És molt important distingir entre crear un nou procés i crear un nou programa.

- Els processos són les entitats, els "subjectes" que empra el sistema operatiu per a planificar els "programes". Dit d'una altra manera: pel sistema operatiu existeixen processos, independentment de l'aplicació que estiguin executant.[1]
- Executar una aplicació vol dir que un determinat procés ha d'executar el codi contingut en un arxiu en un format executable (com per exemple ELF), o un *script*. És a dir, indica què ha de fer un determinat procés, les instruccions que ha d'executar.

Fins ara aquest capítol s'ha centrat sobre processos, i s'ha vist com al crear un nou procés el que es fa és un "duplicat". En aquest apartat es veurà com es pot indicar quin programa, quina aplicació, ha d'executar un procés.

1.5.1 La funció *execve*

Per a executar un nou programa es disposa de la funció `execve`, declarada de la següent manera:

```
#include <unistd.h>

int execve (const char *pathname, char *const argv[],
            char *const envp[]);
```

on `pathname` és la ruta (absoluta o relativa) del fitxer que conté el programa a executar, `argv` és la llista d'arguments (igual al paràmetre `argv` de la funció `main`) i `envp` és la llista de les variables de l'entorn (com per exemple `TERM=xterm`).

[1] Estrictament parlant, això no és cert, ja que els threads també són entitats, subjectes, des del punt de vista del sistema operatiu.

Quan no hi ha error, el programa que venia executant el procés es descarta, i passa a executar el nou procés. Els segments de text (codi), pila i *heap* se substitueixen pels del nou programa. És a dir, tota la memòria, tot l'entorn d'execució del procés, es descarta. Ara bé, el PID del procés es manté.

Una *curiositat* de la funció `execve` és que, si no hi ha error, no retorna mai. El motiu és senzill d'entendre: com que s'executa un nou programa i l'antic ja no existeix, no hi ha cap lloc a on retornar. Si `execve` retorna, no cal comprovar el seu valor de retorn: sempre serà -1, perquè algun error hi haurà hagut.

1.5.2 Exemple per a *execve*

L'exemple següent està tret de [Kerrisk, 2010, pàgina 566]. Consisteix en dos codis font. El primer d'ells és el programa que crida la funció `execve`. El segon és el codi font del programa que s'executarà.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int
main (int argc, char *argv[])
{
  char *argVec[10];
  char *envVec[] = {
    "GREET=salut", "BYE=adieu", NULL
  };

  if (!argv[1])
    return -1;

  argVec[0] = strrchr (argv[1], '/');
  if (argVec[0])
    argVec[0]++;
  else
    argVec[0] = argv[1];
  argVec[1] = "hello world";
  argVec[2] = "goodbye";
  argVec[3] = NULL;

  execve (argv[1], argVec, envVec);
  printf ("execve: %s (%d)\n", strerror (errno), errno);
}
```

```
#include <stdio.h>

extern char **environ;

```

```
5  int
6  main (int argc, char *argv[])
7  {
8    int i;
9    char **ep;
10
11   for (i = 0; i < argc; i++)
12     printf ("argv[%d] = %s\n", i, argv[i]);
13
14   for (ep = environ; *ep != NULL; ep++)
15     printf ("environ: %s\n", *ep);
16
17   return 0;
18 }
```

1.5.3 La família de funcions *exec*

Rares vegades s'empra la funció `execve` directament per a executar un nou programa. La llibreria de C ofereix unes funcions, que internament empren `execve`, que ofereixen algunes comoditats al programador. Aquestes funcions són:

- `execle`
- `execlp`
- `execvp`
- `execv`
- `execl`

Abans de veure com estan declarades, convé explicar quines diferències presenten entre elles:

- La funció `execve` demana que se li especifiqui la ruta del programa, ja sigui absoluta (que comença per `/`) o relativa. Ara bé, quan un usuari especifica ordres en una shell, per posar un exemple, acostuma a especificar les comandes (la immensa majoria de les quals són programes) sense indicar exactament on resideixen. El shell té una llista de directoris a on buscar aquestes comandes, aquesta llista està a la variable d'entorn `PATH`. Per a més detalls sobre aquesta variable d'entorn es pot consultar [Kerrisk, 2010, apartat 27.2.1, pàgines 568–570].

  La funció `execve` no fa servir aquesta llista, però les funcions `execlp` i `execvp` sí la fan servir. De manera que si s'empra aquesta funció per executar el programa (per exemple) `ls`, no cal indicar que aquest programa es troba (habitualment) a `/bin/ls`. Aquestes dues funcions contenen la lletra `p` al seu nom.

- La funció `execve` espera que els arguments per línia de comandes es passin com un array de cadenes. Les funcions `execl`, `execle` i `execlp`, en canvi, reben aquesta llista com una llista d'arguments, un rere l'altre, finalitzada amb un punter `NULL`. Aquestes tres funcions contenen la lletra `l` al seu nom, mentre que la resta contenen la lletra `v`.
- Les funcions `execve` i `execle` permeten especificar una llista de variables d'entorn (d'aquí que tinguin la lletra `e` al seu nom), mentre que la resta de funcions empren, com a variables d'entorn, les mateixes que les del procés que crida la funció.

La declaració d'aquestes funcions és la següent:

```
#include <unistd.h>

int execle (const char *pathname, const char *arg, ...
            /*, (char *) NULL, char *const envp[] */);
int execlp (const char *filename, const char *arg, ...
            /*, (char *) NULL */);
int execvp (const char *filename, char *const argv[]);
int execv (const char *pathname, char *const argv[]);
int execl (const char *pathname, const char *arg, ...
           /*, (char *) NULL */);
```

Totes aquestes funcions només retornen si hi ha error (i quan retornen sempre retornen -1).

1.5.4 Exemple

Com a exemple es presenta una funció, `spawn`, que combina la funcionalitat de `fork` i `execvp` per tal d'executar un nou programa.

```
1  #include <assert.h>
2  #include <errno.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <sys/types.h>
6  #include <sys/wait.h>
7  #include <unistd.h>
8
9  pid_t
10 spawn (char **args)
11 {
12   assert (args);
13
14   pid_t pid = fork ();
15   switch (pid)
```

```
16      {
17      case -1:
18        fprintf (stderr, "fork: %s (%d)\n", strerror (errno), errno);
19        return -1;
20
21      case 0:
22        execvp (args[0], args);
23        fprintf (stderr, "execvp %s: %s (%d)\n", args[0], strerror (errno),
24                 errno);
25        _exit (0);
26
27      default:;
28      }
29
30    return pid;
31  }
32
33  int
34  main (void)
35  {
36    char *args[] = { "ls", "-l", "/", NULL };
37
38    pid_t pid = spawn (args);
39    wait (NULL);
40    printf ("Child's PID was %d\n", pid);
41  }
```


Capítol 2

Sistema bàsic d'entrada i sortida

2.1 Introducció

L'objectiu d'aquest capítol és introduir l'alumne en els conceptes, eines i mecanismes bàsics del sistema d'entrada del sistema operatiu UNIX. El concepte bàsic que s'introduirà serà el *file descriptor*, l'ús del qual s'ampliarà en els posteriors capítols de comunicació entre processos. Posteriorment es veuran els mecanismes elementals per a treballar concretament amb arxius i directoris. Al capítol 10 es presentaran eines i conceptes addicionals, a part de completar el material exposat en aquest capítol 2.

2.2 Eines bàsiques d'entrada i sortida

2.2.1 File descriptor: conceptes

En UNIX totes les crides al sistema que treballen amb arxius utilitzen el que s'anomena *file descriptor* per a referir-se als arxius oberts. Els file descriptors són nombres no negatius, normalment petits. A més de permetre treballar amb arxius, els file descriptors també s'utilitzen per a una multitud de mecanismes de comunicació entre processos, com per exemple comunicacions per Internet, i també per a representar dispositius que no són arxius (com ara terminals, modems, etc).

Per defecte tot procés té oberts, quan comença la seva execució, els següents file descriptors:

1. El descriptor 0 (zero), també anomenat `STDIN_FILENO`, és l'*entrada estàndar*, i generalment representa el teclat. Les funcions de la llibreria estàndar de C que no reben cap `FILE *` i que llegeixen dades (per exemple, `scanf`, `gets`, etc) llegeixen d'aquest file descriptor. La llibreria estàndar de C proveeix un `FILE *` anomenat `stdin` per a l'entrada estàndar.

2. El descriptor 1, també anomenat `STDOUT_FILENO`, és la *sortida estàndar*, i generalment representa la pantalla. Les funcions de la llibreria estàndar de C que no reben cap `FILE *` i que escriuen dades (per exemple, `printf`, `puts`, etc) escriuen en aquest file descriptor. La llibreria estàndar de C proveeix un `FILE *` anomenat `stdout` per a la sortida estàndar.

3. El descriptor 2, també anomenat `STDERR_FILENO`, és la *sortida d'error*, i generalment també representa la pantalla. La llibreria estàndar de C proveeix un `FILE *` anomenat `stderr` per a la sortida d'error. Aquest descriptor és útil per quan volem separar els missatges normals dels missatges d'error, especialment si la sortida estàndar ha estat redirigida (per exemple a un fitxer).

2.2.2 Duplicar file descriptors

Pot ser interessant, per diferents motius, duplicar file descriptors. Per a això es disposa de les funcions següents:

```
#include <unistd.h>

int dup (int oldfd);
int dup2 (int oldfd, int newfd);
```

- La funció `dup` crea (i retorna) un nou file descriptor, que és un duplicat del que se li passa com a argument. Aquest nou file descriptor serà el més petit que estigui lliure.

- La funció `dup2` crea un nou file descriptor, que és un duplicat del que se li passa com a argument al paràmetre `oldfd`. El valor del nou file descriptor és `newfd`, i si aquest file descriptor ja estava obert, es tanca prèviament (ignorant qualsevol error que hi pugués haver en tancar-lo).

2.2.3 Obertura i tancament d'arxius: *open* i *close*

Per a obrir un arxiu es disposa de la funció `open`:

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

int open (const char *path, int flags);
```


Aquesta funció obre l'arxiu anomenat `path` (que pot ser una ruta absoluta, és a dir, començant per `/`, o bé una ruta relativa, entesa a partir del *current working directory* (vegi's l'apartat 2.4.3) del procés, que per defecte és el directori des d'on s'ha executat el programa). El paràmetre `flags` indica quines operacions es podran dur a terme sobre l'arxiu. En primer lloc hi ha tres constants que indiquen quin tipus d'operacions es volen dur a terme amb l'arxiu a obrir:

`O_RDONLY` L'arxiu serà obert només per a llegir.

`O_WRONLY` L'arxiu serà obert només per a escriure.

`O_RDWR` L'arxiu serà obert per a llegir i escriure.

S'ha d'especificar *un i només un* de les tres constants anteriors. A més d'aquestes, i unint-les amb l'operador `|` (*or* binària), es poden especificar més constants:

`O_APPEND` Obliga que totes les escriptures a l'arxiu es facin sempre al final de l'arxiu. Cal que s'hagi especificat la constant `O_WRONLY` o `O_RDWR`.

`O_CREAT` Si l'arxiu a obrir no existeix, la funció `open` no falla sinó que crea l'arxiu (buit). Llavors la funció `open` rep un paràmetre extra, que indica els permisos de l'arxiu (per a més informació sobre els permisos d'un arxiu, vegi's l'apartat 10.3.3). L'efecte d'aquesta constant és independent de si l'arxiu s'obre per lectura o per escriptura.

`O_TRUNC` Si l'arxiu ja existeix, s'elimina prèviament el seu contingut. En Linux aquesta constant també té efecte si l'arxiu només s'ha obert per a lectura, però llavors l'usuari ha de tenir permís d'escriptura sobre l'arxiu.

`O_EXCL` Aquesta constant s'empra conjuntament amb `O_CREAT`, i fa que `open` falli si l'arxiu a obrir ja existeix. La comprovació de l'existència i la creació de l'arxiu es realitzen *atòmicament*, és a dir, que les dues operacions no es poden interrompre (és a dir, que es fan les dues o no se'n fa cap).

`O_NOFOLLOW` No se segueixen els enllaços simbòlics (per a més informació sobre què és un enllaç simbòlic, vegi's l'apartat 2.4.1). Per defecte, `open` *segueix* els enllaços simbòlics, però si aquesta constant està especificada i `path` és un enllaç simbòlic, la funció `open` falla.

`O_SYNC` Obre l'arxiu per a entrada i sortida síncrona. Vegi's l'apartat **??** per a més informació. En alguns UNIX s'anomena `O_FSYNC`.

`O_DIRECT` Fa que el sistema operatiu se salti el *buffer* per tal d'escriure a l'arxiu. Vegi's l'apartat **??** per a més informació.

A part d'aquestes constants, n'hi ha més que es poden consultar a la pàgina manual de la funció `open` (secció 2). Les constants `O_DIRECT` i `O_NOFOLLOW` no estan declarades si no s'ha definit la macro `_GNU_SOURCE` abans d'incloure `fcntl.h`:

```
#define _GNU_SOURCE
#include <fcntl.h>
```

Si la funció `open` ha aconseguit obrir l'arxiu, retorna el seu file descriptor (si un arxiu està obert per diferents processos, cadascun d'ells tindrà el seu propi file descriptor). Si la funció `open` falla, retorna -1 i a la variable global `errno` hi haurà un codi d'error que indicarà el motiu; la pàgina manual de la funció `open` conté una descripció dels codis d'error possibles. També es pot emprar la funció `perror` perquè es mostri un missatge indicant la causa de l'error. La funció `open`, a l'hora de *seleccionar* un file descriptor per a l'arxiu que s'obre, sempre tria el més petit que estigui lliure.

Finalment, la funció `close` rep un sol argument (un file descriptor) i serveix per a tancar un arxiu.

2.2.4 Lectura i escriptura amb un sol buffer: *read* i *write*

Per a realitzar lectures i escriptures a un arxiu es disposa de les dues funcions següents:

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

ssize_t read (int fd, void *buf, size_t nbytes);
ssize_t write (int fd, const void *buf, size_t nbytes);
```

- La funció `read` llegeix del file descriptor indicat per `fd`, llegeix com a molt `nbytes` i els ubica a la zona de memòria apuntada per `buf`. La funció `read` retorna la quantitat real de bytes que s'han aconseguit llegir (si es llegeix d'un arxiu i `read` retorna zero, vol dir que ja s'ha llegit tot l'arxiu). De la zona de memòria apuntada per `buf` només es modifiquen els bytes corresponents a la lectura realitzada, la resta es deixa intacta.

  Important: la funció `read` no posa cap `'\0'` al final del buffer ni al final de les dades llegides.

- La funció `write` escriu al file descriptor indicat per `fd`, escriu com a molt `nbytes` que estan a la zona de memòria apuntada per `buf`. La funció `write` retorna la quantitat real de bytes que s'han aconseguit escriure. La zona de memòria apuntada per `buf` queda intacta.

Si hi ha hagut algun error al llegir o a l'escriure, es retorna `-1` i, a l'igual que amb la funció `open`, es pot emprar la variable `errno` o la funció `perror` per a saber i/o mostrar la causa de l'error.

Què pot fer que `read` o `write` retornin un valor inferior a `nbytes`?

1. La funció `read` retorna `nbytes` si a l'arxiu realment hi ha, almenys, `nbytes` per a llegir. Si la funció `read` s'aplica a d'altres tipus de file descriptors (com ara connexions TCP, *pipes*, o terminals), hi ha d'altres circumstàncies que s'explicaran a l'explicar aquests mecanismes de comunicació entre processos (per exemple, quan la funció `read` llegeix d'un terminal, normalment llegeix línies senceres).

2. La funció `write`, quan es treballa amb arxius, acostuma a retornar sempre `nbytes`; en cas contrari, pot ser que s'hagin sobrepassat límits en la mida d'un arxiu, o algun límit referit al procés, o bé que no quedi espai en disc. Per a d'altres tipus de file descriptors hi ha diferents circumstàncies que poden provocar una *escriptura parcial*.

Exemple Com a exemple, es presenta un programa molt senzill que rep dos arxius com a arguments: el primer arxiu ha d'existir, i el segon es convertirà en una còpia del primer.

```
1  #include <sys/types.h>
2  #include <fcntl.h>
3  #include <stdio.h>
4  #include <unistd.h>
5
6  int
7  main (int argc, char *argv[])
8  {
9    int fd_in, fd_out;
10   char buff[512];
11   ssize_t counter;
12
13   if (argc < 3)
14     {
15       fprintf (stderr, "Falten arguments\n");
16       return -1;
17     }
18
19   if ((fd_in = open (argv[1], O_RDONLY)) < 0)
20     {
21       perror (argv[1]);
22       return -2;
23     }
24
25   if ((fd_out = creat (argv[2], 0644)) < 0)
26     {
27       perror (argv[2]);
28       return -3;
29     }
30
31   do
```

```
32      {
33        counter = read (fd_in, buff, 512);
34        write (fd_out, buff, counter);
35      }
36    while (counter > 0);
37
38    close (fd_in);
39    close (fd_out);
40    return 0;
41  }
```

2.3 Operacions sobre arxius

2.3.1 Creació d'arxius nous: *creat*

A l'apartat 2.2.3 s'ha vist la funció `open` per a obrir (i, en segons quins casos, crear) arxius. Hi ha a més a més la funció `creat`, que actualment es considera obsoleta (ja que les seves funcionalitats queden englobades dins de la funció `open`):

```
#include <fcntl.h>

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

La crida a la funció `creat` és equivalent a la següent crida a `open`:

```
fd = open (pathname, O_WRONLY|O_CREAT|O_TRUNC, mode);
```

La funció `creat`, a l'hora de *seleccionar* un file descriptor per a l'arxiu que es crea, sempre tria el més petit que estigui lliure.

2.3.2 Eliminació d'arxius

Hi ha dues funcions per a eliminar arxius: `unlink` i `remove`:

```
#include <unistd.h>

int unlink (const char *pathname);

#include <stdio.h>

int remove (const char *pathname);
```

La primera només elimina arxius, i la segona elimina arxius i directoris (buits): quan es tracta d'eliminar un arxiu, `remove` crida `unlink`. Hi ha dos detalls sobre l'eliminació d'arxius que cal tenir en compte:

1. Un arxiu no s'elimina fins que tots els processos que el tenien obert l'han tancat. Sovint, quan un programa crea un arxiu temporal, crida la funció d'eliminar-lo immediatament després de crear-lo, per tal d'assegurar-se que un cop acabat de fer servir, desapareixerà.

2. En UNIX existeix el que es coneix amb el nom d'*enllaç dur* (o simplement *enllaç*, vegi's l'apartat 2.4.1 per a més informació). Un enllaç dur ve a ser com un nom per a l'arxiu, de forma que un mateix arxiu pot tenir diversos noms, en diversos directoris d'una mateixa partició. En UNIX, cada arxiu queda únivocament identificat pel seu número d'inode i per la partició on resideix.

   Un arxiu no s'elimina realment fins que s'eliminen tots els seus enllaços durs.

2.3.3 Canviar l'offset d'un arxiu: *lseek*

A l'apartat 10.2.2 s'ha comentat que cada file descriptor corresponent a un arxiu té associat un *offset* que indica en quin punt s'ha de fer la següent lectura o escriptura. Si es desitja modificar aquest *offset* sense haver de fer lectures innecessàries, es pot fer ús de la següent funció:

```
#include <unistd.h>

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

- Si es vol avançar o retrocedir l'*offset* de l'arxiu una quantitat determinada de bytes, el paràmetre `whence` valdrà `SEEK_CUR` i `offset` valdrà la quantitat de bytes que es vol avançar o retrocedir l'*offset*.

- Si es vol establir l'*offset* a una posició determinada, el paràmetre `whence` valdrà `SEEK_SET` i `offset` serà la posició exacta a on ha de quedar l'*offset*.

- Si es vol establir l'*offset* més enllà del final de l'arxiu, el paràmetre `whence` valdrà `SEEK_END` i `offset` serà la quantitat de bytes més enllà del final al qual quedarà posicionat l'*offset*. El byte zero representa el primer byte *després* del final de l'arxiu.

La funció retorna el nou *offset* de l'arxiu. Per tal de saber l'*offset* sense modificar-lo, es pot cridar `lseek` de la següent manera:

```
off_t actual = lseek (fd, 0, SEEK_CUR);
```

Evidentment, la funció `lseek` exigeix que el file descriptor correspongui a un arxiu (és a dir, no es pot aplicar sobre file descriptors corresponents a connexions TCP, etc).

2.4 Enllaços i directoris

2.4.1 Concepte de directori i d'enllaç

Després de parlar, en general, del sistema d'entrada i sortida en UNIX, i de com fer operacions amb arxius, ara és el torn dels enllaços i els directoris. Ja s'ha dit, anteriorment, que en UNIX *tot són arxius*, i els directoris no en són cap excepció: un directori, en UNIX, és un arxiu que relaciona, bàsicament, números d'inode (un inode és un número que identifica, únivocament, un arxiu dins d'una partició o dispositiu) amb noms d'arxiu. Dit d'una altra manera: un directori és un arxiu que posa noms als altres arxius (incloent els directoris).

Concepte d'enllaç dur En diferents apartats s'ha parlat del concepte d'enllaç dur (*hard link*). Un enllaç dur no és res més que la combinació entre un inode i un nom. Des d'aquest punt de vista, un directori és una taula d'enllaços durs. Cada arxiu té, almenys, un enllaç dur, ja que si té un nom és que el seu inode apareix en algun directori. Si el seu inode apareix en més d'un directori, llavors té més d'un enllaç dur. Les metadades d'un arxiu es guarden en una estructura de dades que també rep el nom d'inode; aquest inode conté el número d'enllaços durs que té l'arxiu.

Tots els noms (enllaços durs) d'un arxiu són equivalents i no n'hi ha cap que tingui prioritat sobre la resta. Si un arxiu té un dos noms (a i b) i una aplicació modifica a, llavors b també queda modificat, ja que realment són el mateix arxiu.

Concepte d'enllaç simbòlic A banda dels enllaços durs, en UNIX també hi ha el que es coneix amb el nom d'*enllaç simbòlic*. S'entendrà millor el seu concepte si es compara amb un enllaç dur:

- Un enllaç dur, per ell mateix, no és un arxiu, sinó un altre nom per a un arxiu. En canvi, un enllaç simbòlic és un altre arxiu (per tant, amb el seu número d'inode), el contingut del qual és el nom de l'arxiu al qual apunta.
- Un enllaç dur posa nom a un inode, i només es diferencia dels altres enllaços durs pel nom. Un enllaç simbòlic no té cap mena de relació amb l'inode de l'arxiu que referencia, sinó amb el seu nom. Si aquest arxiu *perd* el nom referenciat per l'enllaç simbòlic (ja sigui perquè canvia de nom o perquè s'elimina l'enllaç dur), l'enllaç simbòlic queda *trencat*.
- Com que els inodes són específics d'un sistema d'arxius i d'una partició, un enllaç dur no pot *creuar* particions (en una partició x no es pot crear un enllaç dur a un arxiu que resideix en una altra partició y). En canvi, com que els enllaços simbòlics no referencien inodes, sí poden *creuar* particions.
- Hi ha implementacions de UNIX, incloent Linux, que impedeixen als usuaris fer enllaços durs de directoris. En general, les implementacions que sí permeten crear enllaços durs a directoris, ho reserven únicament al superusuari.

```
struct dirent {
  ino_t          d_ino;       /* inode number */
  off_t          d_off;       /* offset to the next dirent */
  unsigned short d_reclen;    /* length of this record */
  unsigned char  d_type;      /* type of file */
  char           d_name[256]; /* filename */
};
```

Figura 2.1: `struct dirent`

- Un enllaç dur és completament *resistent* als canvis d'ubicació d'un arxiu. Un enllaç simbòlic no sempre ho serà, ja que el seu contingut és la ruta de l'arxiu. Aquesta ruta pot ser absoluta (és a dir, comença pel directori arrel) o relativa.

2.4.2 Lectura de directoris

En UNIX es disposa de dos mecanismes per a la lectura de directoris. El primer d'aquests mecanismes permet llegir les entrades de directori una a una; el segon, les llegeix totes de cop i les ubica a memòria.

2.4.2.1 Lectura entrada a entrada

El primer pas que cal fer per a llegir un directori és obrir-lo, amb la funció següent:

```
#include <dirent.h>

DIR * opendir (const char *pathname);
```

on `pathname` és el nom del directori que es vol obrir. Si l'obertura no s'ha realitzat correctament, `opendir` retorna un punter `NULL`. Un cop s'ha obert el directori, es pot procedir a llegir les entrades de directori amb la funció següent:

```
#include <dirent.h>

struct dirent * readdir (DIR *dirp);
```

Aquesta funció llegeix, cada cop que se la crida, una entrada de directori. Ella s'encarrega de recordar quina és l'última entrada que s'ha llegit (és una funció amb *memòria interna*).

Quins camps conté un `struct dirent`? Cada implementació de UNIX defineix els següents, però almenys n'ha de contenir dos: el número d'inode i el nom de l'arxiu. La figura 2.1 mostra el contingut de `struct dirent`. La funció `readdir` retorna un punter `NULL` quan no hi ha més entrades de directori per llegir (és a dir, després d'haver llegit l'última).

Per a tancar el directori, es disposa de la funció següent:

```
#include <dirent.h>

int closedir (DIR *dirp);
```

Exemple Com a exemple, es presenta un programa que recorre una jerarquia de directoris, comptant quants arxius hi ha de cada tipus d'arxiu diferent.

```
#include <dirent.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

int isfifo = 0, ischar = 0, isdir = 0, isblock = 0, isreg = 0,
  islink = 0, issocket = 0, total;

static int examina (const int);

int
main (int argc, char *argv[])
{
  char *dirname;
  int current_dir;

  if (!argv[1])
    {
      dirname = malloc (sizeof (char) * 2);
      dirname[0] = '.';
      dirname[1] = 0;
    }
  else
    dirname = argv[1];

  current_dir = open (".", O_RDONLY);
  if (current_dir == -1)
    {
      perror (".");
      return -1;
    }

  examina (current_dir);
  total = isfifo + ischar + isdir + isblock + isreg + islink + issocket;
  printf ("Total:           %d, %3.2f %%\n", total, 100.0);
  printf ("Regulars:        %d, %3.2f %%\n", isreg, isreg * 100.0 / total);
  printf ("Directoris:      %d, %3.2f %%\n", isdir, isdir * 100.0 / total);
  printf ("Disp. car cter:  %d, %3.2f %%\n", ischar, ischar * 100.0 / total);
  printf ("Disp. bloc:      %d, %3.2f %%\n",
          isblock, isblock * 100.0 / total);
```

```
43    printf ("Ennla  simb lic: %d, %3.2f %%\n", islink, islink * 100.0 / total);
44    printf ("UNIX sockets:    %d, %3.2f %%\n",
45            issocket, issocket * 100.0 / total);
46    return 0;
47  }
48
49  int
50  examina (const int fd_dir)
51  {
52    DIR *dirp;
53    struct dirent *entry;
54    int fd;
55
56    if (fchdir (fd_dir) < 0)
57      {
58        perror ("fchdir");
59        return -1;
60      }
61    close (fd_dir);
62
63    if ((dirp = opendir (".")) == NULL)
64      {
65        perror ("opendir");
66        return -2;
67      }
68
69    while ((entry = readdir (dirp)) != NULL)
70      {
71        switch (entry->d_type)
72          {
73          case DT_FIFO:
74            isfifo++;
75            break;
76          case DT_CHR:
77            ischar++;
78            break;
79          case DT_BLK:
80            isblock++;
81            break;
82          case DT_REG:
83            isreg++;
84            break;
85          case DT_LNK:
86            islink++;
87            break;
88          case DT_SOCK:
89            issocket++;
90            break;
91          case DT_DIR:
```

```
92            isdir++;
93            if ((fd = open (entry->d_name, O_RDONLY)) < 0)
94              {
95                perror (entry->d_name);
96              }
97            else
98              {
99                if ((strcmp (entry->d_name, ".") != 0) &&
100                   (strcmp (entry->d_name, "..") != 0))
101                 {
102                   examina (fd);
103                   chdir ("..");
104                 }
105             }
106           break;
107         }
108     }
109
110   return 0;
111 }
```

2.4.2.2 *Funció* scandir

La funció `scandir` permet llegir d'un sol cop les entrades d'un directori, ubicant-les a memòria:

```
#include <dirent.h>

int scandir (const char *path,
             struct dirent ***punter,
             int (*selecciona) (const struct dirent *),
             int (*ordena) (const struct dirent **,
                            const struct dirent **));
```

La funció `scandir` llegeix les entrades del directori `path`, ubicant-les a la zona de memòria apuntada per `punter` (el programador no cal que demani memòria per a aquest array, la funció `scandir` se n'encarrega). Abans d'incloure una entrada de directori a l'array `punter`, `scandir` passa l'entrada a la funció apuntada per `selecciona`; si aquesta funció retorna zero, llavors l'entrada no s'inclou a l'array (si es passa un punter `NULL` com a argument al paràmetre `selecciona`, llavors totes les entrades de directori s'inclouen a l'array). A més a més, les entrades de directori es poden ordenar segons el criteri de la funció `ordena`. Si no es vol fer cap tipus d'ordenació, es pot passar un punter `NULL`; hi ha una funció ja implementada i declarada, de nom `alphasort`, que ordena les entrades alfabèticament.

La funció retorna la quantitat d'entrades de directori guardades a l'array apuntat per `punter`.

Exemple L'exemple que es presenta llegeix totes les entrades de directori excepte per a . i . .

```
#include <dirent.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int
triar (const struct dirent *arg)
{
  if (strcmp (arg->d_name, ".") == 0 || strcmp (arg->d_name, "..") == 0)
    return 0;
  return 1;
}

int
main (int argc, char *argv[])
{
  if (argc < 2)
    {
      fprintf (stderr, "Falten arguments\n");
      return -1;
    }

  struct dirent **arxius;
  int q_arxius = scandir (argv[1], &arxius, triar, alphasort);
  if (arxius == NULL)
    perror ("scandir");
  printf ("Hi ha %d entrades de directori\n", q_arxius);
  while (q_arxius--)
    {
      printf ("[%d] %s\n", q_arxius, arxius[q_arxius]->d_name);
      free (arxius[q_arxius]);
    }
  free (arxius);

  return 0;
}
```

Important: convé fixar-se en com s'allibera la memòria ubicada a l'array.

2.4.3 Concepte de *current working directory*

Per a cada procés hi ha el que es coneix com a *current working directory*. Suposi's el cas que un procés obre un determinat arxiu, com per exemple:

```
open ("myfile", O_RDONLY);
```

Queda el dubte de saber en quin directori està aquest arxiu `myfile`. Si la ruta fos *absoluta* (és a dir, que comencés a partir del directori arrel), aquest dubte no existiria. En el cas de l'exemple, en què la ruta és *relativa*, es fa servir el que es coneix com a *current working directory*. Quan un procés s'inicia, el *current working directory* és el directori des d'on s'ha cridat el procés. Per tal de canviar aquest directori hi ha la funció `chdir`:

```
#include <unistd.h>

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

on `path` és un directori. Per tal de conèixer l'actual *current working directory* hi ha les dues funcions següents:

```
#include <unistd.h>

char *getcwd (char *buffer, size_t size);
char *get_current_dir_name (void);
```

La funció `getcwd` és estàndar de UNIX, mentre que la segona funció és exclusiva de Linux. La funció `getcwd` espera que el programador passi, com a argument al paràmetre `buffer`, una zona de memòria suficientment gran per tal d'encabre-hi la ruta absoluta del *current working directory*, i com a argument a `size` hi passarà la mida de la zona de memòria. La funció `getcwd` posa un caràcter `'\0'` al final del *current working directory*. Si la zona de memòria no és prou gran, llavors `getcdw` retorna `NULL`. En Linux, es pot passar un punter `NULL` a `buffer` (i zero a `size`) i llavors la pròpia funció s'encarrega d'ubicar memòria suficient (l'adreça de la qual és el valor de retorn), i el programador s'ha d'encarregar d'alliberar la memòria ubicada.

La funció `get_current_dir_name` ve a ser equivalent a cridar `getcdw` passant un punter `NULL` i un argument zero al paràmetre `size`. Per tal que aquesta funció estigui declarada, cal haver definit la macro `_GNU_SOURCE` abans d'incloure `unistd.h`.

2.5 La funció *select*

La funció `select` no realitza, per ella mateixa, cap operació d'entrada o sortida, però intervé de manera directa i determinant en les operacions d'entrada i sortida que realitzen moltes aplicacions. Per aquest motiu s'ha decidit incloure-la en aquest capítol.

2.5.1 Plantejament del problema

Suposem una aplicació que vol llegir de diferents file descriptors (per exemple, d'un terminal i d'una connexió a internet) de forma simultània. Davant d'aquesta situació

hi ha diferents alternatives (que es presenten amb més detall al capítol ??). La funció `select` permet monitoritzar diferents file descriptors:

1. Permet monitoritzar file descriptors tant per lectura com per escriptura.[1]
2. Permet especificar un timeout per a la monitorització, no especificar-ne cap (en tal cas el procés queda bloquejat idefinidament fins que algun file descriptor està a punt per a llegir o escriure) o bé fer una consulta "instantània" (un timeout igual a zero).

Els tres arguments més importants que rep la funció `select` són:

- Un punter a un conjunt de file descriptors que serà monitoritzat per lectura. Si es passa un punter `NULL` llavors no es monitoritzarà cap file descriptor per a lectura.
- Un punter a un conjunt de file descriptors que serà monitoritzat per escriptura. Si es passa un punter `NULL` llavors no es monitoritzarà cap file descriptor per a escriptura.
- Un punter a un objecte que definirà el tipus de timeout que es desitja. Aquest objecte té tipus `struct timeval` i indica un temps en segons i microsegons. Hi ha tres alternatives:
  1. Passar un punter `NULL`: no hi ha cap mena de timeout, és a dir, el procés queda bloquejat fins que almenys un file descriptor pot realitzar l'operació indicada (lectura o escriptura).
  2. Especificar un timeout igual a zero segons i zero microsegons: es fa una consulta "instantània" de l'estat dels file descriptors, sense cap mena d'espera.
  3. Especificar un timeout en segons i/o microsegons. Si durant aquest temps algun file descriptor està a punt, llavors la funció `select` retorna sense esperar que hagi passat el temps de timeout. Ara bé, la funció `select` retorna passat aquest timeout, independentment de com estiguin els file descriptors que ha de monitoritzar.

2.5.2 Prototipus

La funció `select` està declarada de la següent manera:

```
#include <sys/time.h> /* En Linux no cal */
#include <sys/select.h>
```

[1] [Kerrisk, 2010, pàgina 131] explica que la funció *select* pot monioritzar dues condicions més, poc freqüents i que no s'explicaran en aquesta obra.

```
int select (int nfds, fd_set *read_fds, fd_set *write_fds,
            fd_set *except_fds, struct timeval *timeout);
```

La funció retorna la quantitat de file descriptors que estan a punt (zero si no n'hi ha cap), i -1 en cas d'error (el més típic és la interrupció per una crida al sistema; per a més detalls consulteu els capítols de senyals).

El significat de cada argument és el següent:

- L'argument `nfds` ha de valdre la suma d'1 més el file descriptor més alt que monitoritzi `select`. Per exemple, si `select` ha de monitoritzar els file descriptors 3, 5, 12 i 69, llavors `nfds` ha de valdre 70 com a mínim.

  L'única finalitat d'aquest argument és permetre que el sistema operatiu sigui més eficient. No hi ha cap problema en passar un valor "exageradament" alt, com ara 512 o 1024.

- L'argument `read_fds` és un punter a un conjunt de file descriptors que seran monitoritzats per lectura — es pot passar `NULL` si no es vol monitoritzar cap file descriptor per lectura. Quan `select` retorna, en aquest conjunt només hi queden els file descriptors que estan preparats per a lectura — és a dir, que si es llegeix d'ells el procés no quedarà bloquejat.

- L'argument `write_fds` té un significat equivalent al de `read_fds`, però monitoritza per escriptura.

- L'argument `except_fds` monitoritza file descriptors per a unes condicions "excepcionals", com ara l'arribada de dades urgents (*out-of-band*) en una connexió TCP. En aquest llibre no se'n parlarà.

- L'argument `timeout` serveix per a indicar un timeout, tal com s'ha explicat anteriorment. El tipus `struct timeval` té dos camps (els dos són enters):

  - `tv_sec`, que indica el número de segons.
  - `tv_usec`, que indica el número de microsegons.

  Si es vol un timeout de dos segons i mig, llavors `tv_sec=2` i `tv_usec=500000`.

  Nota important: els estàndars de UNIX com ara POSIX i SUS indiquen que la funció `select` no ha de modificar l'objecte apuntat per `timeout` quan `select` retorna. A la pràctica, Linux se salta aquest requeriment: quan `select` retorna, es pot consultar `timeout` per saber quant de temps quedava per tal que el timeout expirés. És important tenir en compte aquest fet quan es vol emprar `select` dins d'un bucle.

2.5.3 El tipus *fd_set*

El tipus `fd_set` representa, com el seu nom indica, un conjunt de file descriptors (màxim 1024). Hi ha una sèrie de macros per a operar amb aquest tipus:

`FD_ZERO` Inicialitza el conjunt a zero:

```
fd_set set;
FD_ZERO (&set);
```

`FD_SET` Afegeix un file descriptor a un conjunt:

```
int fd;
fd_set set;
FD_SET (fd, &set);
```

`FD_CLR` Elimina un file descriptor d'un conjunt:

```
int fd;
fd_set set;
FD_CLR (fd, &set);
```

`FD_ISSET` Retorna cert si un file descriptor pertany a un conjunt:

```
int fd;
fd_set set;
if (FD_ISSET (fd, &set)) {}
```

2.5.4 Exemple

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

int
main (void)
{
  fd_set rfds;
  struct timeval tv;
  int retval;

  /* Watch stdin (fd 0) to see when it has input. */
  FD_ZERO (&rfds);
  FD_SET (0, &rfds);

  /* Wait up to five seconds. */
  tv.tv_sec = 5;
```

```
19    tv.tv_usec = 0;
20
21    retval = select (1, &rfds, NULL, NULL, &tv);
22    /* Don't rely on the value of tv now! */
23
24    if (retval == -1)
25      perror ("select()");
26    else if (retval)
27      printf ("Data is available now.\n");
28    /* FD_ISSET(0, &rfds) will be true. */
29    else
30      printf ("No data within five seconds.\n");
31
32    return 0;
33  }
```

Capítol 3

Senyals i temporitzadors: eines bàsiques

3.1 Introducció

Els senyals són un mecanisme que empra el nucli del sistema operatiu per a comunicar events als processos, malgrat que a vegades s'empren com a mecanisme de comunicació entre processos. Com s'anirà veient al llarg d'aquest capítol i del capítol 11, la programació dels senyals és molt més complexa i difícil del que pot semblar. Per tant, es recomana que abans de començar a escriure codi relacionat amb senyals, es llegeixi atentament el material que aquí s'exposa, per tal d'intentar evitar errors tant de disseny com de programació.

Aquest primer capítol servirà per explicar els conceptes bàsics associats als senyals, com enviar senyals i les tècniques més bàsiques per a rebre senyals — concretament els *signal handlers*. La resta de mecanismes s'explicaran al capítol 11, que inclouen mecanismes més avançats per als signal handlers i la recepció de senyals per file descriptors, entre d'altres. En l'exposició del material s'intentarà començar pels conceptes fonamentals i anar progressant de les eines més senzilles fins a les més complexes.

3.2 Conceptes

Un senyal és la notificació d'un event. Cada senyal té un nom associat, que descriu el tipus d'event que notifica. Internament, cada senyal té un número associat, però el programador ha d'usar sempre els noms, ja que els números canvien de sistema operatiu en sistema operatiu, mentre que els noms no canvien. El programador no

Senyal	Acció per defecte	Descripció
SIGABRT	Finalitzar i fer core dumped.	Avortar l'execució d'un procés.
SIGALRM	Finalitzar.	Ha passat una determinada quantitat de temps.
SIGBUS	Finalitzar i fer core dumped.	Error en un accés a memòria.
SIGCHLD	Ignorar.	Un procés fill ha finalitzat o aturat la seva execució.
SIGCONT	Continuar l'execució.	Continuar l'execució d'un procés que estava aturat.
SIGFPE	Finalitzar i fer core dumped.	Error de coma flotant (per exemple, divisió per zero).
SIGHUP	Finalitzar.	Desconnexió del terminal.
SIGILL	Finalitzar.	Intent d'executar una instrucció il·legal.
SIGINT	Finalitzar.	Interrupció de l'execució (Ctrl+C).
SIGKILL	Finalitzar. No es pot capturar.	Assassinar el procés.
SIGPIPE	Finalitzar.	Escriure en una pipe de la qual ningú llegeix.
SIGQUIT	Finalitzar i fer core dumped.	Sortida del terminal.
SIGSEGV	Finalitzar i fer core dumped.	Error en un accés a memòria (segmentation fault).
SIGSTOP	Aturar l'execució. No es pot capturar.	Aturar (però no finalitzar) l'execució del procés.
SIGTERM	Finalitzar.	Finalitzar l'execució d'un procés.
SIGUSR1	Finalitzar.	Senyal d'usuari número 1.
SIGUSR2	Finalitzar.	Senyal d'usuari número 2.

Taula 3.1: Taula-resum de senyals típics de UNIX

es pot inventar nous senyals a part dels que ja hi hagi en el sistema en concret sobre el qual programi.

La taula 3.1 mostra, molt sintèticament, una llista típica (no completa) de senyals en UNIX. No deixa de ser útil, però, donar una mica més de detalls sobre cadascun d'ells, per tal d'il·lustrar quina mena d'events notifiquen els senyals, com se'n serveix el sistema operatiu i com se'n pot servir l'usuari o administrador d'un sistema.

`SIGABRT` Aquest és el senyal que un procés rep quan crida la funció `abort`. Per defecte, aquest senyal fa que el procés finalitzi i generi un *core dump*. Cal tenir en compte que, malgrat que un senyal, per defecte, generi un *core*, és possible que a la pràctica no es creï el fitxer, perquè la seva creació estigui anul·lada o limitada pel *shell*. Vegeu la pàgina manual de `ulimit` per a més detalls.

`SIGALRM` Aquest és el senyal que un procés rep quan expira un temporitzador. Les funcions `alarm` i `setitimer` (explicades en aquest mateix capítol i al capítol 11, respectivament) s'empren per establir temporitzadors. El senyal `SIGALRM` representa temporitzadors de "temps real", és a dir, un temporitzador que compta el temps tal com el percep l'ésser humà.

Amb la funció `setitimer` es pot especificar un temporitzador que només compti el temps que el procés consumeix (és a dir, es compta el pas del temps únicament quan s'executa aquell procés); llavors el senyal que es rep quan expira el temporitzador és `SIGPROF`. Aquest temporitzador compta el temps quan el procés s'executa en mode usuari i en mode kernel.

Quan el temporitzador només compta el temps quan el procés s'executa en mode usuari, llavors el senyal que es rep és `SIGVTALRM`.

`SIGBUS` Aquest senyal indica certs tipus d'errors en accedir a memòria, malgrat que en general, els errors relacionats amb l'accés a memòria provoquen el senyal `SIGSEGV`.

`SIGCHLD` Aquest senyal indica a un procés que algun dels seus fills ha finalitzat l'execució, independentment del motiu. Aquest senyal té algunes normes especials pel que fa a alguns aspectes relacionats amb el funcionament general dels senyals.

`SIGCONT` Quan un procés aturat (amb el senyal `SIGSTOP` o similar) rep aquest senyal, continua la seva execució. Hi ha normes especials que alteren el comportament d'aquest senyal.

`SIGFPE` Aquest senyal indica que s'ha produït un error aritmètic, com per exemple una divisió per zero. De totes maneres, cal tenir present que cada arquitectura decideix quines operacions generen una excepció que provoca la recepció de `SIGFPE`. Per exemple, en arquitectures Intel de 32 bits, una divisió entera per zero sempre provoca una excepció, però si es tracta d'una divisió de nombres decimals (*floating point*), llavors pot no produir-se l'excepció i el resultat seria la representació d'infinit. Per a més informació, es pot consultar la pàgina manual de `fenv`.

`SIGHUP` Estrictament, aquest senyal significa que el terminal s'ha desconnectat; en aquesta obra no es parlarà de terminals. De totes formes, aquest senyal és important perquè és el que fan servir molts dimonis per tornar a llegir l'arxiu

de configuració. És a dir, enviar el senyal `SIGHUP` a un dimoni fa que (la majoria d'ells, no tots) torni a llegir l'arxiu de configuració.

`SIGILL` Aquest senyal el reben els processos que intenten executar una instrucció *assembler* que no existeix o està mal formada.

`SIGINT` Aquest és el senyal que rep un procés quan es prem la combinació de tecles Ctrl+C.

`SIGKILL` Aquest és el senyal que s'utilitza per a matar un procés de forma definitiva, ja que aquest senyal no es pot ignorar, bloquejar o capturar amb un *signal handler*.

Quan un usuari vol finalitar un procés, convé no emprar, com a primer recurs, aquest senyal, sinó `SIGTERM`. Com que `SIGKILL` no es pot capturar, el procés no té l'oportunitat d'alliberar els recursos que estigui emprant en aquell moment, i això és especialment important per aquells recursos que no s'alliberen automàticament amb la mort d'un procés (com ara certs tipus de semàfors i de memòria compartida, per exemple, o arxius temporals).

`SIGPIPE` Aquest és el senyal que un procés rep quan intenta escriure en una *pipe*, una FIFO o un socket que no tenen cap procés lector, és a dir, que el sistema operatiu sap que les dades que s'escriuen no les llegirà ningú.

`SIGQUIT` Aquest senyal és similar a `SIGINT`, però a més a més fa que el procés generi un *core*.

`SIGSEGV` Aquest senyal tan popular s'envia a un procés que intenta accedir a una zona de memòria sense tenir permís, ja sigui perquè no està mapejada (o perquè només hi pot accedir el kernel) o perquè intenta escriure en una zona que només es pot llegir.

`SIGSTOP` Aquest senyal fa que un procés aturi la seva execució. Hi ha d'altres senyals que també provoquen que un procés aturi la seva execució, però el senyal `SIGSTOP` no es pot ignorar, bloquejar o capturar amb un *signal handler*.

`SIGTERM` Aquest senyal sol·licita a un procés que finalitzi la seva execució. Aquest senyal es pot ignorar, bloquejar o capturar amb un *signal handler*. Aquest senyal dóna l'oportunitat al procés d'alliberar els recursos que usés i finalitzi la seva execució de forma "neta" (per exemple, eliminant els arxius temporals que estigués utilitzant).

`SIGUSR1` i `SIGUSR2` Aquests dos senyals estan reservats específicament per als programadors, és a dir, el sistema operatiu no enviarà mai cap d'aquests dos senyals. La resta de senyals que s'han vist estan pensats per a ser enviats pel sistema operatiu, o bé en circumstàncies molt especials. Tant `SIGUSR1` com `SIGUSR2` estan disponibles per ser emprats per qualsevol motiu que el programador consideri útil.

Cal tenir en compte que l'acció per defecte d'aquests dos senyals és finalitzar l'execució del procés.

3.3 Enviament de senyals

3.3.1 Funció *kill*

Un procés pot enviar senyals a un altre procés. A tal fi, es disposa de la funció `kill`, que està declarada de la següent manera:

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

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

on `pid` és un enter que identifica el procés al qual es vol enviar el senyal, i `sig` indica el nom (o nombre) del senyal. Enviar el senyal o (zero) a un procés significa comprovar si existeix. Un procés només pot enviar senyals a un altre procés si es compleix almenys una de les dues condicions següents (tot i que el senyal `SIGCONT` té unes regles especials per a ell):

1. O bé l'usuari que executa el procés que envia el senyal és el superusuari (el superusuari és l'administrador en un sistema UNIX).

2. O bé l'usuari efectiu del procés que envia el senyal coincideix amb el del procés que el rep.

El concepte d'*usuari efectiu* és el següent: si l'usuari `user` executa un programa, per defecte l'usuari efectiu del procés és `user`. Ara bé, pot ser que el programa tingui un bit activat que es diu `SETUID`; en aquest cas, l'usuari efectiu del procés no és el de l'usuari que executa el programa, sinó el del propietari. Per exemple, si el programa pertany a l'usuari `root` i el bit `SETUID` està activat, l'usuari efectiu serà `root`, encara que qui hagi executat el programa sigui `user`.

3.3.2 Funció *raise*

Per tal que un procés s'enviï un senyal a ell mateix, es disposa de la funció `raise`:

```
#include <signal.h>

int raise (int sig);
```

on el paràmetre `sig` és el senyal a enviar. En un procés amb múltiples threads, `raise` envia el senyal al thread que ha cridat aquesta funció.

3.3.3 Comentaris addicionals

L'enviament de senyals no funciona com una cua. És a dir, per a cada procés, el sistema operatiu guarda un conjunt (*màscara*) de senyals pendents. Un senyal pot pertànyer al conjunt o no, però si hi pertany, només hi pertany un cop. Si un senyal està bloquejat i, mentre està bloquejat, es rep diverses vegades, llavors quan el senyal es desbloquegi s'entregarà un sol cop, no tantes vegades com hagués arribat.

De fet, no cal ni tan sols que el senyal estigui bloquejat perquè passi el que s'ha comentat. Pot ser que un senyal arribi múltiples vegades abans que el sistema operatiu tingui temps d'entregar-lo al procés. En aquest cas el senyal també s'entregarà un sol cop al procés. El lector interessat pot consultar uns quants exemples a [Kerrisk, 2010, apartat 20.12].

3.4 Processos i senyals

Per a cada procés, el sistema operatiu guarda:

- El conjunt de senyals pendents de ser entregat al procés. Un senyal pot pertànyer o no al conjunt, i si hi pertany, hi pertany un sol cop.
- El conjunt de senyals bloquejats (també anomenat màscara de senyals). Un senyal pot pertànyer o no al conjunt, i si hi pertany, hi pertany un sol cop. Un senyal bloquejat és aquell senyal que el sistema operatiu entregarà al procés un cop aquest procés el desbloquegi.

Pel que fa als senyals, el procés pot establir-ne la seva disposició:

- Pot ignorar el senyal, de manera que mai s'entregarà al procés. Es poden ignorar tots els senyals excepte dos: `SIGKILL` i `SIGSTOP`. Quan s'ignora un senyal, l'execució del procés continua de forma normal, excepte per als senyals generats per excepcions de hardware (com ara `SIGSEGV`); en aquest cas, el resultat és *undefined behaviour*.
- Pot establir l'acció per defecte — a vegades és ignorar, a vegades és finalitzar el procés.
- Pot establir un *signal handler*, és a dir, una funció que s'executarà un cop s'entregui el senyal.

La disposició d'un senyal és "paral·lela" al fet de si està bloquejat o no. Per exemple, es pot tenir un senyal bloquejat i instal·lar-hi un signal handler: mentre el senyal estigui bloquejat, no s'executarà el signal handler encara que arribi el senyal corresponent.

Per a canviar la disposició d'un senyal hi ha la funció `signal` (que s'explicarà a l'apartat 3.5) i la funció `sigaction` (que s'explicarà al capítol 11).

3.5 La funció *signal*

Aquesta funció permet canviar la disposició d'un senyal:

- Permet especificar que s'ignori el senyal.
- Permet especificar que s'executi l'acció per defecte a l'arribar un senyal.
- Permet instal·lar un signal handler per a un senyal.

La funció `signal` està declarada de la següent manera:

```
#include <signal.h>

typedef void (*sig_t)(int);

sig_t signal (int sig, sig_t func);
```

on `sig` és el senyal, i `func` indica la disposició del senyal.

- Si `func` val `SIG_IGN`, el senyal passa a estar ignorat.
- Si `func` val `SIG_DFL`, s'activa l'acció per defecte.
- Si `func` no val ni `SIG_IGN` ni `SIG_DFL`, llavors ha de ser el nom d'una funció que rep un `int` (que serà el senyal que ha provocat l'execució del signal handler) i retorna `void`.

El valor de retorn de la funció `signal` és l'anterior disposició del senyal, o el signal handler que hi hagués instal·lat.

Exemple

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

static void sig_usr (int);

int
main (void)
{
  signal (SIGUSR1, sig_usr);
  signal (SIGUSR2, sig_usr);

  while (1)
    pause ();
```

```
  return 0;
}

static void
sig_usr (int signo)
{
  switch (signo)
    {
    case SIGUSR1:
      printf ("SIGUSR1 received.\n");
      break;

    case SIGUSR2:
      printf ("SIGUSR2 received.\n");
      break;

    default:
      printf ("Signal number %d received.\n", signo);
      break;
    }
}
```

3.6 Signal handlers: introducció

Un *signal handler* és una funció que s'executa quan arriba un determinat senyal. El sistema operatiu és l'encarregat d'executar el signal handler, i a tal fi interromp l'execució del procés. Per tant, el programador no pot preveure en quin moment de l'execució el procés aquesta es veurà interrompuda per l'execució d'un signal handler. Quan finalitza l'execució del signal handler, el sistema operatiu segueix executant el procés allà on l'havia interromput.

En teoria, un *signal handler* pot fer pràcticament qualsevol cosa; a la pràctica, però, pel fet d'interrompre el flux d'execució, és molt recomanable que es limitin a fer el mínim imprescindible, i que siguin tan simples com sigui possible.

3.6.1 Entrega de senyals

Si el senyal (per exemple) `SIGUSR1` arriba i té instal·lat un signal handler, llavors s'executa aquest signal handler, i durant la seva execució el senyal `SIGUSR1` queda bloquejat. Per tant, si tornés a arribar `SIGUSR1`, el signal handler seguiria executant-se, i al finalitzar, es tornaria a executar perquè ha tornat a arribar `SIGUSR1`. Ara bé, com que els senyals no s'encuen, si durant l'execució del signal handler el senyal `SIGUSR1` arriba diversos cops, el signal handler es tornarà a executar *un sol cop*: un senyal pot pertànyer o no al conjunt de senyals pendents, però si hi pertany, hi pertany només un sol cop.

Ara bé, suposem que el signal handler és compartit entre `SIGUSR1` i `SIGUSR2`. Llavors, si durant l'execució del signal handler arriba `SIGUSR2`, llavors l'execució

del signal handler s'interrmp i es torna a iniciar, i quan retorna, segueix l'execució del signal handler allà on havia quedat quan va arribar `SIGUSR2`.

Suposem el cas concret de `SIGCHLD`, el senyal que arriba quan un procés fill mor. Si aquest senyal té associat un signal handler, llavors què passa amb els fills que es moren durant l'execució del signal handler? Si se'n mor un, podem estar tranquils perquè un cop finalitzi l'execució del signal handler, aquest es tornarà a executar. Però i si se'n mor més d'un? Llavors si el signal handler està mal dissenyat, hi haurà processos zombie, ja que el signal handler s'executarà *un sol cop* més, i no tants cops més com fills han mort.

3.6.2 Funcions reentrants

En paraules de l'estàndar *Single UNIX Specification*, versió, 3, una funció reentrant és aquella

> whose effect, when called by two or more threads, is guaranteed to be as if the threads each executed the function one after the other in an undefined order, even if the actual execution is interleaved.

Per *thread* no només cal entendre els threads que s'expliquen a la tercera part del llibre (que vénen a ser com processos, però amb molts més recursos compartits, com ara l'espai d'adreces), sinó genèricament "fluxos d'execució". El procés és un flux d'execució, i el signal handler en seria un altre, pel fet de poder interrompre el flux d'execució del procés *en qualsevol moment*.

Quina importància té que una funció sigui reentrant?

- Suposem una funció que només accedeix a variables automàtiques (o sigui, variables locals que no estan declarades amb la paraula clau `static`). Si aquesta funció és interrompuda i es torna a executar, la segona execució no interferirà amb la primera, ja que les variables automàtiques "viuen" a la pila, i per cada crida a la funció hi haurà una còpia de les variables automàtiques. Per tant, es resultats seran correctes — suposant que aquesta funció només crida funcions reentrants, o no crida cap altra funció.

- Suposem que la funció accedeix a variables estàtiques (variables globals, o variables locals declarades amb la paraula clau `static`). Si aquesta funció és interrompuda i es torna a a xecutar, la segona pot interferir amb la primera. Imaginem que accedeix a una estructura de dades global, com ara una llista. La primera execució insteria un element a la llista, i a mitja inserció s'interromp i es torna a executar. La segona execució es troba amb l'element a mig inserir, i pot ser que algun punter seguís a `NULL`, etc. És molt probable és que el procés mori per *segmentation fault*.

Hi ha moltes funcions que no són reentrants, i el lector interessat pot trobar-ne una llista a [Kerrisk, 2010, pàgina 426]. Com a exemple pràctic de cas en què un signal handler crida una funció no reentrant, es presenta el següent exemple:

```
1  #include <pwd.h>
2  #include <signal.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <unistd.h>
6
7  static void
8  my_alarm (int signo __attribute__ ((unused)))
9  {
10   struct passwd *rootptr;
11   printf ("In %s\n", __func__);
12
13   if ((rootptr = getpwnam ("root")) == NULL)
14     perror ("getpwnam (\"root\");");
15   alarm (1);
16 }
17
18 int
19 main (void)
20 {
21   struct passwd *ptr;
22   signal (SIGALRM, my_alarm);
23   alarm (1);
24
25   while (1)
26     {
27       if ((ptr = getpwnam ("daemon")) == NULL)
28         perror ("getpwnam(\"daemon\");");
29       else if (strcmp (ptr->pw_name, "daemon") != 0)
30         printf ("Return value corrupted!, " "pw_name = %s\n",
31                 ptr->pw_name);
32     }
33
34   return 0;
35 }
```

Aquest programa fa que, cada segon, s'executi el *signal handler*. Si el lector compila i executa aquest programa, observarà que finalitza al cap d'unes iteracions amb un *segmentation fault*. El motiu és que la funció `getpwnam` no és reentrant. És a dir, mentre s'executa la funció `getpwnam` no es pot tornar a cridar aquesta mateixa funció. Per això es diu que no és reentrant: no permet que s'hi torni a entrar mentre s'és a dins.

3.6.3 Accés a variables globals

És força típic que un signal handler es limiti a modificar una variable global, que posteriorment es comprova per decidir si cal fer una acció o una altra. No és cap

problema que un signal handler accedeixi a una variable global, sempre i quan es prenguin les precaucions que es mencionen en aquest apartat.

1. La variable global s'ha de declarar amb el qualificador `volatile`, per evitar certes optimitzacions que podrien resultar molt perilloses. Al capítol 11 es presentarà un cas pràctic on s'il·lustra què pot passar si no se segueix aquest consell.

2. Els accessos a les variables globals no sempre són atòmics. Per això hi ha el tipus `sig_atomic_t`, en què les assignacions (només les assignacions) són atòmiques. És a dir, és impossible interrompre una operació atòmica, per tant, l'assignació no es podrà interrompre.

3.7 Bloqueig de senyals

Tot i que en aquest capítol no veurem cap eina que requereixi que un senyal estigui bloquejat, bloquejar un senyal és tan senzill que val la pena explicar-ho en aquest capítol, i així alleugerir la càrrera del capítol 11.

Abans d'entrar en detalls convé explicar el tipus `sigset_t`. Aquest tipus ve a ser un conjunt o màscara de senyals, i el programador disposa de les següents funcions per a treballar-hi:

```
#include <signal.h>

int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signum);
int sigdelset (sigset_t *set, int signum);
int sigismember (const sigset_t *set, int signum);
```

- La funció `sigemptyset` inicialitza el conjunt apuntat per `set`, fent que no contingui cap senyal.

- La funció `sigfillset` inicialitza el conjunt apuntat per `set`, fent que contingui tots els senyals.

- La funció `sigaddset` afegeix el senyal `signum` al conjunt apuntat per `set`.

- La funció `sigdelset` elimina el senyal `signum` del conjunt apuntat per `set`.

- La funció `sigismember` retorna cert si el senyal `signum` pertany al conjunt apuntat per `set`.

Ara ja podem presentar la funció següent:

```
#include <signal.h>

int sigprocmask (int how,
                 const sigset_t *restrict set,
                 sigset_t *oset);
```

Aquesta funció altera la màscara de senyals (o sigui, el conjunt de senyals bloquejats) d'un procés:

- Si es vol afegir un senyal a la màscara de senyals, llavors `how` val `SIG_BLOCK` i `set` conté únicament el senyal que es vol afegir al conjunt de senyals bloquejats.
- Si es vol desbloquejar un senyal prèviament bloquejat, llavors `how` val `SIG_UNBLOCK` i `set` conté únicament el senyal que es vol eliminar del conjunt de senyals bloquejats.
- Si es vol substituir el conjunt de senyals bloquejats, llavors `how` val `SIG_SETMASK` i `set` conté exactament els senyals que estaran bloquejats.

En qualsevol dels tres casos, si `oset` no val `NULL`, la funció `sigprocmask` farà que apunti a l'antiga màscara de senyals.

Cal recordar que els senyals `SIGKILL` i `SIGSTOP` no es poden bloquejar.

Capítol 4

Temporitzadors en UNIX

4.1 Introducció

Moltes aplicacions requereixen dur a terme accions concretes amb el pas del temps, o amb una determinada freqüència; en d'altres casos pot ser necessari o convenient mesurar el pas del temps. En aquest capítol s'exposaran alguns dels mecanismes més emprats per a establir temportitzadors en Linux.

4.2 Funció *alarm*

El primer temporitzador és la funció `alarm`, declarada de la següent manera:

```
#include <unistd.h>

unsigned int alarm (unsigned int seconds);
```

on `seconds` és el nombre de segons que passaran abans que el procés rebi el senyal `SIGALRM`. Si aquest nombre de segons és zero, es desactiva l'enviament del senyal. El valor de retorn de la funció és:

1. Si no hi havia cap alarma programada, retorna zero.
2. Si hi havia una alarma programada, retorna el nombre de segons que quedaven per tal que s'activés.

Cal tenir en compte que les alarmes programades amb la funció `alarm` no són permanents, és a dir, el senyal es rep un cop (quan passa el temps especificat) i prou.. Si es vol que es torni a activar, cal tornar a cridar la funció `alarm`.

4.3 Funció *setitimer*

La funció `setitimer` és molt semblant a `alarm`, i està declarada de la següent manera:

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

int setitimer (int which,
               const struct itimerval *newvalue,
               struct itimerval *oldvalue);
```

La funció `setitimer` presenta els següents avantatges respecte `alarm`:

1. Permet especificar una resolució molt més alta (en principi fins a microsegons).
2. Permet especificar el temps de repetició, és a dir: un cop produïda la primera alarma, cada quan s'ha de tornar a activar. D'aquesta manera no cal tornar a cridar `setitimer`.
3. Permet especificar com s'ha de comptar el pas del temps.

Rep els següents arguments:

- L'argument `which` indica com s'ha de comptar el pas del temps:

  `ITIMER_REAL` Computa el pas del temps real, tal com el mesuraria qualsevol rellotge. Fa que el procés rebi el senyal `SIGALRM`.

  `ITIMER_VIRTUAL` El temps només avança mentre el procés s'executa. Fa que el procés rebi el senyal `SIGVTALRM`.

  `ITIMER_PROF` El temps avança mentre el procés s'executa i mentre el sistema operatiu realitza accions *en nom* del procés (per exemple, executar una crida al sistema). Fa que el procés rebi el senyal `SIGPROF`.

- L'argument `newval` apunta a l'objecte que definirà la nova alarma que es vol programar.
- L'argument `oldval` apunta a un objecte que contindrà l'alarma que hi havia programada, suposant que n'hi hagués alguna de programada.

L'estructura `struct itimerval` està definida de la següent manera:

```
struct itimerval
{
  struct timeval it_interval;
  struct timeval it_value;
};
```

On el camp `it_value` indica (en segons i microsegons) el temps que ha de passar per tal que es generi l'alarma; el camp `it_interval` indica (també en segons i microsegons) els temps successius als quals s'aniran generant les alarmes. L'estructura `struct timeval` està definida de la següent manera:

```
struct timeval
{
  long tv_sec;
  long tv_usec;
};
```

4.4 Timers per file descriptors

Linux disposa d'un mecanisme propi, que consisteix en crear un file descriptor associat a un temporitzador. D'aquesta manera es poden usar funcions com ara `select` o `epoll` per monitoritzar temporitzadors, a més de ser una manera força senzilla per mantenir, simultàniament, diversos temporitzadors.

4.4.1 Creació del file descriptor

El primer pas és servir-se de la funció `timerfd_create`:

```
#include <sys/timerfd.h>

int timerfd_create(int clockid, int flags);
```

on `clockid` pot ser `CLOCK_REALTIME` o `CLOCK_MONOTONIC` (la diferència principal és que al segon no l'afectaria un hipotètic canvi de l'hora) i `flags` pot ser zero, un dels següents valors o la seva combinació amb l'operador OR:

`TFD_NONBLOCK` El file descriptor passa a ser no bloquejant.

`TFD_CLOEXEC` El file descriptor es tancarà quan el procés executi alguna de les funcions de la família `exec`.

La funció retorna un file descriptor.

4.4.2 Establiment del timer

Per a especificar el valor del timer disposem de la funció següent:

```
#include <sys/timerfd.h>

int timerfd_settime(int fd, int flags,
                    const struct itimerspec *new_value,
                    struct itimerspec *old_value);
```

on el tipus `struct itimerspect` està definit de la següent manera:

```
struct timespec {
  time_t tv_sec;                /* Seconds */
  long   tv_nsec;               /* Nanoseconds */
};

struct itimerspec {
  struct timespec it_interval;  /* Interval for periodic timer */
  struct timespec it_value;     /* Initial expiration */
};
```

Els arguments de la funció `timerfd_settime` són els següents:

`fd` El file descriptor obtingut amb `timerfd_create`.

`flags` Pot ser zero, o pot valdre `TFD_TIMER_ABSTIME`. Indica com s'ha d'interpretar el valor especificat per `new_value`.

`new_value` Valor del temporitzador, en segons i nanosegons. Si `flags` val zero, indica la quantitat de temps que ha de passar, a partir d'ara, per tal que el timer expiri. Si val `TFD_TIMER_ABSTIME`, llavors indica un valor exacte de temps (és a dir, una data i una hora, en format UNIX).

`old_value` Es pot passar un punter `NULL`; en cas contrari, indica el valor anterior de `new_value`.

4.4.3 Lectura del file descriptor

Un cop tenim creat el file descriptor, com sabem que el timer ha expirat? Simplement llegint del file descriptor. Si opera en mode bloquejant (comportament per defecte), `read` bloquejarà el procés fins que el timer expiri; si està en mode no bloquejant, llavors `read` retornarà -1 (codi d'error `EAGAIN`) si el timer no ha expirat.

A la funció `read` se li ha de passar un enter de 64 bits. Quan el timer expira, aquest enter conté el número de vegades que el timer ha expirat des de l'última lectura.

4.4.4 Exemple

El següent exemple està extret de la pàgina manual de `timerfd_create`, i consisteix en una aplicació que crea un temporitzador i en monitoritza el seu progrés. S'accepten fins a tres arguments per línia de comandes:

1. El primer argument especifica el número de segons per al primer cop que el timer ha d'expirar.

2. El segon argument especifica el número de segons de l'intèrval (és a dir, un cop el timer ha expirat el primer cop).

3. El tercer argument especifica la quantitat de vegades que el timer ha d'expirar abans que l'aplicació finalitzi.

Només el primer argument és obligatori, el segon i el tercer són opcionals.

```
#include <sys/timerfd.h>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>                /* Definition of uint64_t */

#define handle_error(msg) \
               do { perror(msg); exit(EXIT_FAILURE); } while (0)

static void
print_elapsed_time (void)
{
  static struct timespec start;
  struct timespec curr;
  static int first_call = 1;
  int secs, nsecs;

  if (first_call)
    {
      first_call = 0;
      if (clock_gettime (CLOCK_MONOTONIC, &start) == -1)
        handle_error ("clock_gettime");
    }

  if (clock_gettime (CLOCK_MONOTONIC, &curr) == -1)
    handle_error ("clock_gettime");

  secs = curr.tv_sec - start.tv_sec;
  nsecs = curr.tv_nsec - start.tv_nsec;
  if (nsecs < 0)
    {
      secs--;
      nsecs += 1000000000;
    }
  printf ("%d.%03d: ", secs, (nsecs + 500000) / 1000000);
}

int
main (int argc, char *argv[])
{
```

```
42    struct itimerspec new_value;
43    int max_exp, fd;
44    struct timespec now;
45    uint64_t exp, tot_exp;
46    ssize_t s;
47
48    if ((argc != 2) && (argc != 4))
49      {
50        fprintf (stderr, "%s init-secs [interval-secs max-exp]\n", argv[0]);
51        exit (EXIT_FAILURE);
52      }
53
54    if (clock_gettime (CLOCK_REALTIME, &now) == -1)
55      handle_error ("clock_gettime");
56
57    /* Create a CLOCK_REALTIME absolute timer with initial
58       expiration and interval as specified in command line */
59
60    new_value.it_value.tv_sec = now.tv_sec + atoi (argv[1]);
61    new_value.it_value.tv_nsec = now.tv_nsec;
62    if (argc == 2)
63      {
64        new_value.it_interval.tv_sec = 0;
65        max_exp = 1;
66      }
67    else
68      {
69        new_value.it_interval.tv_sec = atoi (argv[2]);
70        max_exp = atoi (argv[3]);
71      }
72    new_value.it_interval.tv_nsec = 0;
73
74    fd = timerfd_create (CLOCK_REALTIME, 0);
75    if (fd == -1)
76      handle_error ("timerfd_create");
77
78    if (timerfd_settime (fd, TFD_TIMER_ABSTIME, &new_value, NULL) == -1)
79      handle_error ("timerfd_settime");
80
81    print_elapsed_time ();
82    printf ("timer started\n");
83
84    for (tot_exp = 0; tot_exp < (uint64_t) max_exp;)
85      {
86        s = read (fd, &exp, sizeof (uint64_t));
87        if (s != sizeof (uint64_t))
88          handle_error ("read");
89
90        tot_exp += exp;
```

```
91        print_elapsed_time ();
92        printf ("read: %llu; total=%llu\n",
93                (unsigned long long) exp, (unsigned long long) tot_exp);
94      }
95
96    exit (EXIT_SUCCESS);
97  }
```

4.5 La funció *select*

La funció `select` presenta, en Linux, una particularitat important que pot ser útil. L'últim argument d'aquesta funció és un punter a un objecte de tipus `struct timeval`, que indica el temps màxim d'espera de `select` (si el punter val `NULL`, llavors l'espera és "eterna"). La pregunta que ens podem fer és: si hi ha activitat en un file descriptor abans que aquest temps passi, quin valor conté l'objecte apuntat? Els estàndars de UNIX diuen que no s'ha de modificar, però Linux no ho compleix. Aquest incompliment el podem aprofitar si volem fer una lectura amb un timeout. Per exemple, si el que volem és llegir d'un socket però no volem quedar-nos eternament esperant les dades, podem usar `select` d'una manera similar a la de l'exemple següent:

```
1   #ifndef _GNU_SOURCE
2   #define _GNU_SOURCE
3   #endif
4
5   #include <assert.h>
6   #include <errno.h>
7   #include <stdio.h>
8   #include <sys/select.h>
9   #include <unistd.h>
10
11  ssize_t
12  read_with_to (int fd, size_t max, void *buffer, int to)
13  {
14    assert (fd > 0);
15    assert (buffer);
16    assert (to > 0);
17
18    struct timeval tv = { to, 0 };
19    fd_set set;
20
21    ssize_t tmp, cnt = 0;
22
23    do
24      {
25        FD_ZERO (&set);
26        FD_SET (fd, &set);
27
28        switch (select (FD_SETSIZE, &set, NULL, NULL, &tv))
```

```
29          {
30          case -1:
31            if (errno == EINTR)
32              continue;
33            else
34              {
35                perror ("select");
36                return -1;
37              }
38            break;
39
40          case 0:
41            if (cnt < (ssize_t) max)
42              printf ("Read only %zd bytes out of %zu\n", cnt, max);
43            return cnt;
44          }
45
46        if (FD_ISSET (fd, &set))
47          {
48            tmp = read (fd, buffer + cnt, max - cnt);
49            switch (tmp)
50              {
51              case -1:
52                perror ("read");
53                return -1;
54
55              case 0:
56                printf ("EOF reached\n");
57                return cnt;
58
59              default:
60                cnt += tmp;
61              }
62          }
63      }
64    while (cnt < (ssize_t) max);
65
66    printf ("Remaining %ld seconds %ld microseconds\n", tv.tv_sec, tv.tv_usec);
67    return cnt;
68  }
```

La funció `read_with_to` rep quatre arguments:

`fd` File descriptor del qual s'ha de llegir.

`max` Màxim número de bytes a llegir.

`buffer` Punter a la zona de memòria on s'ubicaran les dades llegides.

`to` Timeout en número de segons.

La funció comença fent comprovacions bàsiques als arguments amb la macro `assert`. A continuació té lloc el bucle de la funció, que es va repetint mentre la quantitat de bytes llegits (variable local `cnt`) sigui inferior al màxim de bytes a llegir. Fixeu-vos que la funció `select` es crida passant, com a últim argument, un punter a la variable `tv`, que s'inicialitza amb el número de segons que l'usuari indica quan crida la funció. A mesura que `select` va retornant, el valor de la variable va disminuïnt.

La primera sentència `switch` comprova el valor de retorn de `select`. Si és -1 és que s'ha produït algun error. Si l'error és que la funció s'ha interromput per un senyal, es torna a cridar; altrament es retorna -1. El valor de retorn de la funció `read` també es comprova amb una sentència `switch`:

- Si és -1 és que s'ha produït un error, la funció retorna -1.
- Si és zero és que no hi ha més bytes a llegir d'aquell file descriptor (si es tracta d'un arxiu, és que ja hem arribat al final; si és una connexió TCP, és que s'ha tancat).
- Altrament, `read` retorna la quantitat de bytes llegits, que sumem al comptador `cnt`.

Val la pena observar com cridem la funció `read`: li passem el file descriptor, li passem el buffer sumant-li la quantitat de bytes llegits fins al moment (si no ho féssim així, sobreescriuríem les dades ja llegides) i, com a últim argument, la quantitat de bytes a llegir menys els que ja hem llegit.

Capítol 5

Pipes, fifos i cues de missatges

5.1 Introducció

Aquest capítol va dedicat a tres mecanismes de comunicació entre processos d'una mateixa màquina: les pipes, les fifos i les cues de missatges. La finalitat de cadascun d'aquestes eines és transmetre bytes entre processos, sense que el programador s'hagi de preocupar de la sincronització entre els processos involucrats. Les pipes permeten enviar un flux de bytes sense intervenció del sistema de fitxers, per la qual cosa només es pot emprar entre processos "relacionats" (pares i fills, per exemple), mentre que les fifos, al fer intervenir el sistema de fitxers (creen una entrada de directori), permeten comunicar dades entre processos que no tenen cap relació. Les cues de missatges s'assemblen a les pipes i les fifos, però introduint el concepte de missatge.

5.2 Pipes

Les pipes són el mecanisme més antic de comunicació de processos que hi ha en unix, i van aparèixer a principis dels anys setanta. Una pipe és un canal de comunicació amb dos file descriptors, un dels quals serveix per a llegir i l'altre per a escriure. Les pipes presenten les següents característiques:

1. Una pipe és un flux de bytes, és a dir, tant és escriure primer deu bytes i després vint que primer vint i després deu. No existeix el concepte de *missatge* o *frontera* entre missatges. El procés que llegeix de la pipe agafa dades en blocs de mida arbitrària, independentment de la mida dels blocs que s'han escrit. El que sí es conserva és l'ordre en què s'han escrit: és a dir, els primers bytes que es llegeixen són els primers que han entrat a la pipe. No és possible cridar

la funció `lseek` en una pipe. Les lectures en una pipe són destructives: és a dir, quan es llegeixen btyes d'una pipe, aquests bytes *surten* i ja no es poden tornar a llegir.

2. Les lectures són bloquejants: si un procés intenta llegir d'una pipe en què no hi ha dades, el procés quedarà bloquejat fins que algú (un altre procés, per exemple) hi escrigui dades. Si la banda d'escriptura de la pipe està tancada (és a dir, no hi ha cap procés que tingui obert un file descriptor per a escriure en aquesta pipe), llavors el procés no es bloquejarà sinó que veurà un final de fitxer — per exemple, la funció `read` retornarà zero.

   Cal tenir en compte que hi ha mecanismes per a establir un file descriptor en mode no-bloquejant; en aquest cas, la funció `read` retornaria -1 i la variable global `errno` valdria `EAGAIN`, indicant que en aquell moment no hi ha dades a la pipe pendents de llegir.

3. Les pipes són unidireccionals. Si un procés escriu dades a una pipe i immediatament després llegeix de la pipe, llegirà les mateixes dades que acaba d'escriure — suposant que, entre la lectura i l'escriptura, cap altre procés ha llegit de la mateixa pipe.

4. Les pipes tenen una capacitat limitada. Des del punt de vista del sistema operatiu, una pipe és simplement un *buffer*, que té una capacitat màxima. Quan una pipe ha arribat a la capacitat màxima, les escriptures quedaran bloquejades fins que algú llegeixi dades de la pipe (i per tant alliberi espai). En Linux, la capacitat d'una pipe solia ser de 4 KB, actualment és de 64 KB. Les versions més noves del nucli de Linux permeten modificar aquesta capacitat, tot i que és difícil pensar per què podria ser necessari.

   Cal tenir en compte que els senyals poden interrompre una escriptura a la pipe, i retornar indicant que s'han escrit menys bytes dels sol·licitats (*escriptura parcial*).

5.3 Creació i ús de pipes

5.3.1 Funció *pipe*

Per a crear una pipe es disposa de la següent funció:

```
#include <unistd.h>

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

Si la pipe s'ha pogut crear, aquesta funció retorna zero; en cas d'error, retorna -1. L'argument `filedes` és un array d'enters, que serà omplert per la pròpia funció `pipe`. El primer element de l'array és el file descriptor de lectura de la pipe, el segon element és el file descriptor d'escriptura.

5.3.2 Ús d'una pipe

Una pipe a penes és útil en un sol procés (a menys que sigui un procés *multi-threaded*, cas que no es considerarà en aquest capítol ni en aquesta primera part de l'obra), la seva utilitat és comunicar processos, concretament processos *relacionats*, és a dir, que comparteixen un *antecessor* comú que ha creat la pipe. Tenint en compte que es tracta d'un mecanisme unidireccional, el més comú és que un dels processos llegeixi de la pipe i l'altre procés escrigui, de manera que el procés lector tanca el file descriptor d'escriptura i el procés escriptor tanca el file descriptor de lectura, com en l'exemple següent:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void
foo (void)
{
  int fd_pipe[2];
  if (pipe (fd_pipe) < 0)
    perror ("pipe");
  else
    switch (fork ())
      {
      case -1:
        perror ("fork");
        return;

      case 0:
        /* child will read from pipe,
         * so it closes the write fd */
        close (fd_pipe[1]);
        exit (0);

      default:
        /* parent will write to pipe,
         * so it closes the read fd */
        close (fd_pipe[0]);
      }
}
```

Cal remarcar la importància de tancar els file descriptors que no s'usen d'una pipe. Si el procés que llegeix d'una pipe no ha tancat el file descriptor d'escriptura, mai sabrà si els processos escriptors han acabat d'escriure a la pipe, perquè la lectura sempre quedarà bloquejada esperant que ell mateix (el propi procés lector) escrigui a la pipe. Pel que fa a un procés escriptor, tanca el descriptor de lectura perquè quan no hi hagi cap altre procés llegint de la pipe, l'escriptura fallarà i llavors el procés sabrà que ningú està llegint de la pipe. Si no tanqués el descriptor de lectura, podria

arribar a omplir del tot la pipe i quedar-se bloquejant esperant que algú (de fet, ell mateix) llegís de la pipe.

Si es desitja implementar una comunicació bidireccional entre dos processos, llavors caldrà crear dues pipes, i tenir compte de no causar un *deadlock* entre ells — per exemple, que un procés estigui esperant que l'altre escrigui en una pipe mentre aquest segon procés justament espera que el primer escrigui en una altra pipe.

A continuació s'ofereix un exemple complet de programa que treballa amb pipes. El procés pare envia al fill la cadena rebuda per línia de comandes, tota d'un sol cop, i el fill la llegeix en fragments i la mostra per pantalla.

```
1  #include <sys/wait.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <unistd.h>
6
7  #define BUF_SIZE 16
8
9  int
10 main (int argc, char *argv[])
11 {
12   if (argc < 2)
13     {
14       fprintf (stderr, "%s: Not enough arguments\n", argv[0]);
15       return -1;
16     }
17
18   int pfd[2];
19   char buf[BUF_SIZE];
20   ssize_t numRead;
21
22   if (pipe (pfd) < 0)
23     {
24       perror ("pipe");
25       return -1;
26     }
27
28   switch (fork ())
29     {
30     case -1:
31       perror ("fork");
32       return -1;
33
34     case 0:
35       close (pfd[1]);
36       while (1)
37         {
38           numRead = read (pfd[0], buf, BUF_SIZE);
```

```
39            switch (numRead)
40              {
41              case -1:
42                perror ("read");
43                _exit (-1);
44
45              case 0:
46                write (1, "\n", 1);
47                _exit (0);
48
49              default:
50                write (1, buf, numRead);
51              }
52          }
53        _exit (0);
54
55      default:
56        close (pfd[0]);
57        write (pfd[1], argv[1], strlen (argv[1]));
58        close (pfd[1]);
59        wait (NULL);
60        exit (0);
61      }
62  }
```

5.4 Lectura i escriptura en pipes

A diferència d'altres sistemes de comunicació com ara les cues de missatges, una pipe és un canal de bytes o no hi ha cap mena de concepte de "missatge"; és a dir, una pipe tracta les dades com un simple flux de bytes — ara bé, sí es preserva l'ordre amb què s'han realitzat les escriptures. El que s'explica en aquest apartat val tant per a les pipes com per a les fifos.

5.4.1 Escriptura

Quan un procés escriu n bytes en una pipe, i assumint que hi ha "algú" (algun procés) que té la pipe oberta per lectura, poden passar dues coses:

1. Que hi hagi espai a la pipe per ubicar-hi els n bytes que s'hi volen escriure. En aquest cas, la funció `write` (o equivalent) escriu els bytes i retorna immediatament, sense bloquejar el procés.

2. Que no hi hagi espai suficient a la pipe per ubicar-hi els n bytes que s'hi volen escriure. En aquest cas cal distingir si el file descriptor és bloquejant (comportament per defecte) o no bloquejant:

a) Si el file descriptor és bloquejant, el procés quedarà adormit fins que algun altre procés hagi llegit de la pipe, i per tant hagi fet prou espai per a encabir-hi els n bytes.

b) Si el file descriptor no és bloquejant, s'escriuran tants bytes com hi càpiga a la pipe, i la funció `write` retornarà la quantitat de bytes que s'han pogut escriure (*partial write*).

   Hi ha una excepció que cal comentar. Si la quantitat de bytes que caben a la pipe és inferior a un mínim determinat, no s'escriurà cap byte i la funció `write` (o equivalent) retornarà -1 i `errno` valdrà `EAGAIN`. Aquest mínim determinat pren un valor que es comentarà en aquest mateix apartat.

3. Suposi's que hi ha diversos processos escrivint alhora en una pipe. Es podria donar el cas que a la pipe es "barregessin" les dades escrites pels diferents processos? Es podria donar la següent successió d'esdeveniments:

   a) Un procés, anomenat A, vol escriure 1 MB a la pipe, però només hi caben 4 KB. El sistema operatiu escriu 4 KB a la pipe i bloqueja el procés fins que quedi espai.

   b) Un altre procés (B) llegeix 4 KB de la pipe, deixant 4 KB lliures.

   c) Un tercer procés (C) escriu 4 KB a la pipe.

   Observi's que les dades dels processos A i B han quedat mesclades. El sistema operatiu no s'espera que A acabi d'escriure totes les dades que volia abans de donar pas a C. Malgrat tot, el sistema operatiu assegura que hi ha un "mínim d'atomicitat", que almenys ha de ser de 512 bytes. És a dir, que el sistema operatiu assegura que les escriptures són atòmiques en blocs de 512 bytes: aquest bloc s'escriu del tot (sense mesclar-lo amb dades d'altres processos), o no s'escriu en absolut. A aquest valor li direm "mínim d'atomicitat", i en Linux sol ser de 4 KB (tot i que es pot canviar).

   És per això que, quan el file descriptor és no bloquejant, si a la pipe hi caben menys bytes que els que marca el mínim d'atomicitat (i es volen escriure més bytes dels que caben a la pipe), no es fa l'escriptura, ja que s'estaria violant el concepte mateix d'aquest mínim.

5.4.2 Lectura

Quan un procés vol llegir n bytes en una pipe, poden passar diferents coses:

1. Que a la pipe hi hagi, almenys, n bytes. En aquest cas, el sistema operatiu entrega al procés els n bytes requerits, de forma immediata.

2. Que a la pipe hi hagi menys de n bytes. Aquí cal fer més distincions:

a) Que no hi hagi cap altre procés que tingui oberta la pipe per a escriure. En aquest cas, el sistema operatiu entrega els bytes que hi hagi a la pipe. Si no n'hi ha, la funció `read` (o equivalent) retorna zero, indicant *end of file*.

b) Que hi hagi algun altre procés que tingui oberta la pipe per a escriure. Novament cal fer més distincions:

   i. Si el file descriptor és bloquejant (comportament per defecte), el procés queda bloquejat fins que a la pipe han arribat (d'un sol cop o successivament) fins a n bytes.

   ii. Si el file descriptor no és bloquejant, el sistema operatiu entrega al procés els bytes que hi hagi, i la funció `read` retorna un valor inferior a n. D'això se'n viu *partial read*.

5.5 FIFOs

El comportament d'una fifo és exactament igual que el d'una pipe, però amb presència al sistema de fitxers (és a dir, una fifo es pot veure a l'executar la comanda `ls`). Per això a les fifos també se les acostuma a anomenar *named pipes*, ja que són pipes amb un nom. De fet, un cop creada la pipe, s'obre igual que si fos un arxiu.

5.5.1 Creació d'una FIFO

El primer pas per utilitzar una fifo és crear-la, mitjançant la funció següent:

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

int mkfifo (const char *pathname, mode_t mode);
```

on `pathname` és el nom de l'*arxiu*, és a dir, el nom que la fifo tindrà al sistema d'arxius, i `mode` és equivalent al mateix argument de la funció `creat` (per quan es vol crear un arxiu).

5.5.2 Obertura d'una FIFO

Un cop la fifo s'ha creat, cal obrir-la. L'obertura d'una fifo també es fa amb la funció `open`, però la semàntica és diferent:

- Quan s'obre la fifo per lectura (`O_RDONLY`), la crida a `open` bloqueja el procés fins que un altre procés obre la mateixa fifo per escriptura.

- Quan s'obre la fifo per escriptura (`O_WRONLY`), la crida a `open` bloqueja el procés fins que un altre procés obre la mateixa fifo per lectura.

- Alguns UNIXs, incloent Linux, es pot evitar aquest comportament bloquejant obrint la fifo per lectura i escriptura alhora (`O_RDWR`). Ara bé, això té els següents inconvenients:

  1. No és un comportament estàndar: en alguns UNIXs funciona, en d'altres no.
  2. El procés que obre la pipe per lectura i escriptura alhora mai veurà un *end of file* (`read` no retornarà mai zero) a la fifo, perquè el mateix file descriptor amb què llegeix de la fifo serveix per escriure-hi (per a més detalls, vegi's l'apartat 5.4).

L'apartat 44.8 (pàgines 909–915) de Kerrisk [2010] conté un exemple complet d'aplicació que utilitza fifos.

5.6 Cues de missatges

Les cues de missatges tenen una finalitat pràcticament idèntica amb les pipes i les fifos: transmetre dades d'una banda a una altra (entre processos d'una mateixa màquina), però tenen una diferència fonamental, que rau en com tracten els bytes que s'hi transmeten:

- Les pipes transmeten bytes. Quan un procés escriu en una pipe, el sistema operatiu copia els bytes en un buffer, i si aquest està ple el procés s'ha d'esperar (excepte si el file descriptor està en mode no bloquejant). No importa com s'agrupin els bytes a l'hora d'escriure o llegir: es poden escriure primer 100 bytes i després 50, i llegir primer 75 bytes i 75 després. Les pipes (i les fifos) són file descriptors, com ja s'ha vist.

- Les cues de missatges transmeten *missatges*, formats òbviament per bytes. Ara bé, els missatges són infragmentables. No es pot llegir part d'un missatge, o llegir uns quants missatges d'un sol cop. A més a més, els missatges tenen tipus o prioritat, i es poden llegir com si fos una cua (*first-in, first-out*) o bé per tipus (o prioritat).

- Tot i que no està directament relacionat amb el funcionament d'una pipe o d'una cua de missatges, és important tenir en compte que les pipes i les fifos són file descriptors, mentre que les cues de missatges no (vegeu l'excepció que s'explica més endavant).

Com passa amb els semàfors i la memòria compartida, hi ha dues implementacions de cues de missatges: la de System V (la més antiga i tradicional) i la de l'estàndar POSIX. Les cues de missatges de System V cal eliminar-les explícitament, ja que el sistema operatiu no les elimina quan finalitza el procés.

En aquest llibre explicarem les dues implementacions: la de System V és la que tradicionalment sempre s'ha explicat; la de POSIX és especialment útil en Linux perquè els seus identificadors són file descriptors (només en Linux).

5.6.1 Creació d'una cua de missatge

5.6.1.1 System V

Per a crear una cua (o obrir-ne una de ja existent) disposem de la funció següent:

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

int msgget (key_t key, int msgflg);
```

on `key` és un valor retornat per la funció `ftok` o és la constant `IPC_PRIVATE`. En el primer cas, pot ser l'identificador d'una cua ja existent o una de nova; en el segon, estem creant una nova cua de missatges. El paràmetre `msgflg` contindrà els permisos de la cua (igual que en la creació d'un arxiu nou), a part de poder combinar els dos valors següents:

`IPC_CREAT` Si no hi ha cap cua de missatge amb identificador `key`, es crea una nova cua de missatges.

`IPC_EXCL` Si s'ha especificat `IPC_CREAT` i ja existeix una cua de missatges amb identificador `key`, la funció `msgget` fallarà.

Exemple El següent exemple crea una cua de missatges i acte seguit la tanca:

```
1  #include <stdio.h>
2  #include <sys/msg.h>
3
4  int
5  main (void)
6  {
7    int q_id = msgget (IPC_PRIVATE, 0600 | IPC_CREAT);
8    if (q_id < 0)
9      {
10       perror ("msgget");
11       return -1;
12     }
13
14   /* Com que el sistema operatiu no elimina les cues
15    * autom ticament, ho hem de fer expl citament. */
16   msgctl (q_id, IPC_RMID, NULL);
17 }
```

5.6.1.2 POSIX

La funció `mq_open` serveix per crear una cua de missatges o obrir-ne una de nova:

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

mqd_t mq_open (const char *name, int oflag,
               /* mode_t mode, struct mq_attr *attr */);
```

L'argument `name` identifica la cua de missatges. En Linux, un nom consisteix en una barra inclinada (`/`) i una cadena de com a molt 251 caràcters (que no poden ser la barra inclinada). El paràmetre `oflag` és molt similar al de la funció `open`. Per exemple, per a crear una nova cua de missatges, especificarem `O_CREAT`.

Si s'obre una cua que ja existeix, llavors `mq_open` només rep dos arguments. Però si es tracta de crear una nova cua, llavors rep dos arguments addicionals: `mode` indica els permisso de la cua de misssatges (igual que al crear un arxiu nou), i `attr` especifica els atributs de la cua de missatges (es pot passar un punter `NULL` perquè la cua es creï amb els atributs per defecte). Vegeu l'apartat 5.6.5 per a més detalls.

Exemple Per a compilar aquest exemple cal passar l'opció `-lrt` al compilador. De fet, totes les funcions de cues de missatges POSIX exigeixen passar aquesta opció al compilador (almenys en Linux i FreeBSD).

```
1  #include <fcntl.h>
2  #include <mqueue.h>
3  #include <stdio.h>
4  #include <sys/stat.h>
5
6  int
7  main (void)
8  {
9    mqd_t cua_id = mq_open ("/myqueue", O_CREAT | O_EXCL | O_RDWR, 0600, NULL);
10   if (cua_id < 0)
11     {
12       perror ("mq_open");
13       return -1;
14     }
15
16   /* La cua est  creada, ara l'eliminem */
17   mq_close (cua_id);
18   mq_unlink ("/myqueue");
19 }
```

5.6.2 Tancar una cua de missatges

5.6.2.1 System V

Per a tancar una cua de missatges emprarem la funció `msgctl`:

```
#include <sys/msg.h>

int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

on `msqid` és l'identificador de la cua retornat per `msgget`, `cmd` és l'operació que es vol realitzar (en aquest cas, `IPC_RMID` i `buf` apunta a una zona de memòria que s'empra per a algunes operacions de control. Per a eliminar una cua, `buf` s'ignora i pot ser un punter `NULL`.

5.6.2.2 POSIX

Quan un procés ja no necessita seguir emprant una cua de missatges, pot cridar la funció següent:

```
#include <mqueue.h>

int mq_close (mqd_t mqdes);
```

on `mqdes` és el descriptor de la cua retornat per `mq_open`. Aquesta funció es crida implícitament quan el procés mor.

Ara bé, per tal d'eliminar definitivament la cua, cal cridar la funció següent:

```
#include <mqueue.h>

int mq_unlink (const char *name);
```

on `name` és l'argument `name` que s'ha passat a `mq_open` quan s'ha creat (o obert) la cua de missatges.

5.6.3 Enviar missatges

5.6.3.1 System V

Per a enviar un missatge es disposa de la funció següent:

```
#include <sys/msg.h>

int msgsnd (int msq_id, const void *msg_ptr size_t msg_size,
            int msg_flags);
```

El significat de cada paràmetre és el següent:

`msq_id` És l'identificador d'una cua de missatges, obtingut amb `msgget`.

`msg_ptr` Apunta a una estructura que conté el missatge (vegeu l'explicació de més endavant).

msg_size Indica la quantitat de bytes que conté el missatge, sense comptar el seu tipus (vegeu l'explicació de més endavant).

msg_flags Pot ser zero o IPC_NOWAIT. En aquest segon cas, l'operació és no bloquejant.

La funció msgsnd envia el missatge del tot o no l'envia en absolut, és a dir, no fragmenta el missatge. Si la cua de missatges està plena i msg_flags val zer, el procés s'adorm fins que hi hagi espai lliure a la cua; si msg_flags val IPC_NOWAIT llavors la funció retorna -1 i el codi d'error és EAGAIN.

Un missatge d'una cua de missatges és una estructura que ha de contenir, almenys, dos camps, com s'observa a continuació:

```
struct buffer
{
  long type;
  char data[msg_size];
};
```

El primer camp, type, és el tipus del missatge, i ha de ser més gran que zero. El segon camp (de fet, els següents camps, ja que n'hi pot haver més d'un), és el missatge pròpiament dit, i la seva mida és la que s'ha d'indicar al paràmetre msg_size.

Exemple

```
1  #include <stdio.h>
2  #include <sys/msg.h>
3
4  struct my_message
5  {
6    long type;
7    char frase[512];
8  };
9
10 int
11 main (void)
12 {
13   int q_id = msgget (IPC_PRIVATE, 0600 | IPC_CREAT);
14   if (q_id < 0)
15     {
16       perror ("msgget");
17       return -1;
18     }
19
20   struct my_message missatge = { 1, "Hello, world!" };
21
22   if (msgsnd (q_id, &missatge, sizeof (missatge) - sizeof (long), 0) < 0)
```



```
23      {
24        perror ("msgsnd");
25        return -1;
26      }
27
28    /* Com que el sistema operatiu no elimina les cues
29     * autom ticament, ho hem de fer expl citament. */
30    msgctl (q_id, IPC_RMID, NULL);
31  }
```

5.6.3.2 POSIX

Per a enviar un missatge es disposa de la funció següent:

```
#include <mqueue.h>

int mq_send (mqd_t mqdes, const char *msg_ptr, size_t msg_len,
             unsigned msg_prio);
```

El significat de cada paràmetre és el següent:

`mqdes` És l'identificador de la cua de missatges, obtingut amb `mq_open`.

`msg_ptr` Apunta al missatge que es vol enviar. No ha de ser necessàriament una cadena de caràcters.

`msg_len` Indica la mida del missatge que no pot ser superior a la mida màxima d'un missatge (vegeu l'apartat 5.6.5).

`msg_prio` Indica la prioritat del missatge. La prioritat zero és la més baixa. Quan un missatge s'afegeix a la cua, es posa darrere de l'últim missatge de la mateixa prioritat.

Està permès enviar missatges de mida zero.

Exemple

```
1   #include <fcntl.h>
2   #include <mqueue.h>
3   #include <stdio.h>
4   #include <sys/stat.h>
5
6   int
7   main (void)
8   {
9     mqd_t cua_id = mq_open ("/myqueue", O_CREAT | O_EXCL | O_RDWR, 0600, NULL);
10    if (cua_id < 0)
11      {
```

```
12        perror ("mq_open");
13        return -1;
14      }
15
16    char msg[] = "Hello, world!";
17    if (mq_send (cua_id, msg, sizeof (msg), 0) < 0)
18      {
19        perror ("mq_send");
20        return -1;
21      }
22
23    mq_close (cua_id);
24    mq_unlink ("/myqueue");
25  }
```

Enviament amb timeout La funció `mq_timedsend` permet especificar un timeout, de manera que si el procés queda bloquejat, es pot limitar fins quan es vol esperar. Per a més detalls, es recomana consultar la pàgina manual corresponent.

5.6.4 Rebre missatges

5.6.4.1 System V

Per a obtenir un missatge d'una cua de missatges es disposa de la funció següent:

```
#include <sys/msg.h>

ssize_t msgrcv (int msq_id, void *msg_ptr, size_t max_size,
                long msg_type, int msg_flags);
```

Aquesta funció llegeix (lectura destructiva, és a dir, un cop el missatge s'ha llegit s'elimina de la cua) un missatge de mida màxima `max_size` (si la mida del missatge és superior, llavors no es fa cap lectura i `msgrcv` retorna -1 amb el codi d'error `E2BIG`, però vegeu el flag `MSG_NOERROR`). Aquesta mida màxima no inclou el primer camp del missatge (que corresponia al seu tipus).

Els missatges no tenen per què llegir-se en el mateix ordre amb què s'han enviat:

- Si `msg_type` és igual a zero, llavors es llegeix el primer missatge de la cua, independentment del seu tipus.
- Si `msg_type` és superior a zero, llavors es llegeix el primer missatge de la cua que tingui el mateix tipus que `msg_type`.
- Si `msg_type` és inferior a zero, llavors es llegeix el primer missatge de la cua que tingui un tipus igual o inferior al valor absolut de `msg_type`.

L'argument `msg_flags` pot valdre zero o una combinació dels següents flags:

`IPC_NOWAIT` Si no hi ha cap missatge que es pugui llegir, llavors `msgrcv` adorm el procés fins que n'hi hagi un; però amb aquest flag `msgrcv` no adorm el procés sinó que retorna -1 amb el codi d'error `ENOMSG`.

`MSG_EXCEPT` Aquest flag és exclusiu de Linux i exigeix que la macro `_GNU_SOURCE` estigui definida abans d'incloure `sys/msg.h`. Només té significat si `msg_type` és superior a zero, i fa que es llegeixi el primer missatge que el seu tipus **no sigui** `mst_type`.

`MSG_NOERROR` Per defecte, si el missatge a llegir té una mida superior a `max_size`, `msgrcv` retorna error. Si aquest flag està especificat, llavors `msgrcv` trunca el missatge (és a dir, llegeix fins a `max_size`), i la part que sobra es perd.

La funció `msgrcv` retorna la quantitat de bytes que s'han llegit del missatge.

Exemple

```
#include <stdio.h>
#include <sys/msg.h>

struct my_message
{
  long type;
  char frase[512];
};

int
main (void)
{
  int q_id = msgget (IPC_PRIVATE, 0600 | IPC_CREAT);
  if (q_id < 0)
    {
      perror ("msgget");
      return -1;
    }

  struct my_message msg = { 0, {0} };

  if (msgrcv (q_id, &msg, 512, 0, IPC_NOWAIT) < 0)
    perror ("msgrcv");

  printf ("Missatge de tipus %ld, missatge: %s\n", msg.type, msg.frase);

  /* Com que el sistema operatiu no elimina les cues
   * autom ticament, ho hem de fer expl citament. */
  msgctl (q_id, IPC_RMID, NULL);
}
```

5.6.4.2 *POSIX*

La funció `mq_receive` obté (lectura destructiva) el missatge més antic de la màxima prioritat:

```
#include <mqueue.h>

ssize_t mq_receive (mqd_t mqdes, char *msg_ptr, size_t msg_len,
                    unsigned *msg_prio);
```

El significat de cada paràmetre és el següent:

`mqdes` És el descriptor de la cua de missatges, obtingut amb la funció `mq_open`.

`msg_ptr` Apunta a la zona de memòria on s'ubicarà el missatge, i ha de tenir espai per encabir-hi `msg_len` bytes.

`msg_len` Indica la mida de la zona de memòria apuntada per `msg\_ptr`, i ha de ser almenys igual a la mida màxima d'un missatge.

`msg_prio` Si no val `NULL`, apunta a una zona de memòria on s'hi guardarà la prioritat del missatge llegit.

Si la cua està buida, o bé el procés es posa a dormir o bé (si està en mode no bloquejant) retorna -1 amb el codi d'error `EAGAIN`.

Exemple

```
1  #include <fcntl.h>
2  #include <mqueue.h>
3  #include <stdio.h>
4  #include <sys/stat.h>
5
6  int
7  main (void)
8  {
9    mqd_t cua_id = mq_open ("/myqueue", O_CREAT | O_EXCL | O_RDWR, 0600, NULL);
10   if (cua_id < 0)
11     {
12       perror ("mq_open");
13       return -1;
14     }
15
16   struct mq_attr attr;
17   mq_getattr (cua_id, &attr);
18   attr.mq_flags = O_NONBLOCK;
19   mq_setattr (cua_id, &attr, NULL);
20
21   char msg[attr.mq_msgsize];
```

```
22    unsigned prio;
23
24    ssize_t sst = mq_receive (cua_id, msg, attr.mq_msgsize, &prio);
25    if (sst < 0)
26      perror ("mq_receive");
27    else
28      printf ("Received message of %ld bytes, priority %u, message: %s\n", sst,
29              prio, msg);
30
31    mq_close (cua_id);
32    mq_unlink ("/myqueue");
33  }
```

Lectura amb timeout La funció `mq_timedreceive` permet especificar un timeout, de manera que si el procés queda bloquejat, es pot limitar fins quan es vol esperar. Per a més detalls, es recomana consultar la pàgina manual corresponent.

5.6.5 Atributs d'una cua de missatges POSIX

El tipus `struct mq_attr` serveix per a especificar (o obtenir) els atributs d'una cua de missatges, i està definit de la següent manera:

```
struct mq_attr
{
  long mq_flags;
  long mq_maxmsg;
  long mq_msgsize;
  long mq_curmsgs;
};
```

El significat de cada camp és el següent:

`mq_flags` Indica els flags amb què s'ha creat la cua de missatges: zero o `O_NONBLOCK`. Aquest camp s'ignora durant la creació d'una cua, però es pot consultar o modificar amb les funcions `mq_getattr` i `mq_setattr`, respectivament.

`mq_maxmsg` Indica la màxima quantitat de missatges (sense llegir) que pot contenir una cua. Es pot especificar al crear la cua, i es pot consultar posteriorment amb la funció `mq_getattr`.

`mq_msgsize` Indica la mida màxima (en bytes) que pot contenir un missatge. Es pot especificar al crear la cua, i es pot consultar posteriorment amb la funció `mq_getattr`.

`mq_curmsgs` Indica la quantitat de missatges sense llegir que actualment hi ha a la cua. Es pot consultar amb la funció `mq_getattr`.

La funció `mq_getattr` permet consultar els atributs d'una cua:

```
#include <mqueue.h>

int mq_getattr (mqd_t mqdes, struct mq_attr *attr);
```

i la funció `mq_setattr` permet especificar els atributs d'una cua:

```
#include <mqeue.h>

int mq_setattr (mqd_t mqdes, const struct mq_attr *new_attrs,
                struct mq_attr *old_attrs);
```

on `new_attrs` apunta als nous atributs, i `old_attrs` apunta als atributs que hi havia abans de cridar `mq_setattr`.

5.6.6 Altres operacions

5.6.6.1 System V

La funció `msgctl`, a més de servir per eliminar una cua de missatges, també serveix per consultar i establir algunes propietats relacionades amb les cues de missatges. Aquesta funció, que ja s'ha vist a l'apartat 5.6.2, està decladara de la següent manera:

```
#include <sys/msg.h>

int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

on `cmd`, a més de `IPC_RMID` pot valdre `IPC_STAT` (per a consultar els atributs) o `IPC_SET` (per a establir els atributs). El tipus `struct msqid_ds` està definit de la següent manera:

```
struct msqid_ds
{
  struct ipc_perm msg_perms;
  time_t msg_stime;
  time_t msg_rtime;
  time_t msg_ctime;
  unsigned long __msg_cbytes;
  msgqnum_t msq_qnum;
  msglen_t msg_qbytes;
  pid_t msg_lspid;
  pid_t msg_lrpid;
};

struct ipc_perm
```



```
{
  key_t __key;
  uid_t uid;
  gid_d gid;
  uid_t cuid;
  uid_t cgid;
  unsigned short mode;
  unsigned short __seq;
}
```

El significat de cada camp és el següent:

msg_perm Permisos de la cua de missatges.

msg_stime Data i hora de l'última crida a `msgsnd`.

msg_rtime Data i hora de l'última crida a `msgrcv`.

msg_ctime Data i hora de l'última operació `IPC_SET`.

__msg_cbytes Número de bytes continguts en els missatges pendents a la cua.

msg_qnum Número de missatges pendents a la cua.

msg_qbytes Màxim valor per a `__msg_cbytes`, és a dir, límit de bytes que pot contenir la cua.

msg_lspid PID de l'últim procés que ha escrit un missatge a la cua.

msg_lrpid PID de l'últim procés que ha llegit un missatge de la cua.

5.6.6.2 POSIX

Aquesta implementació de les cues de missatges disposa d'un mecanisme de notificació, que permet avisar de forma asíncrona en el moment que una cua buida passa a tenir almenys un missatge. A tal fi es disposa de la funció següent:

```
#include <mqueue.h>

int mq_notify (mqd_t mqdes, const struct sigevent *ptr);
```

Abans d'entrar en detalls, cal tenir en compte els següents aspectes:

1. Només un sol procés pot estar registrat per a rebre notificacions. Si ja hi ha un altre procés registrat, `mq_notify` retorna -1 amb el codi d'error `EBUSY`.
2. Només es notifica quan una cua deixa d'estar buida. Si al moment de registrar-se la cua tingués missatges, no es rebrà cap notificació fins que la cua es buidi i rebi un nou missatge.

3. Les notificacions no són permanents: un cop el procés ha rebut una notificació, automàticament es "des-registra".

4. Si algun procés està bloquejat en una crida a `mq_receive`, llavors serà aquest procés el que rebrà el nou missatge, i el procés registrat no rebrà cap notificació — però seguirà registrat.

5. Un procés es pot "des-registrar" cridant `mq_notify` passant un punter `NULL` com a argument a `ptr`.

El tipus `struct sigevent` està declarat de la següent manera:

```
struct sigevent
{
  int sigev_notify;
  int sigev_signo;
  union sigval sigev_value;
  void (*sigev_notify_function) (union sigval);
  void *sigev_notify_attributes;
};

union sigval
{
  int sival_int;
  void *sival_ptr;
};
```

L'element `sigev_notify` és el que s'empra per determinar com s'ha d'entregar la notificació al procés:

`SIGEV_NONE` Es registra el procés per a la notificació, però quan un missatge arriba quan la cua és buida, en realitat no es fa cap notificació al procés.

`SIGEV_SIGNAL` Es registra el procés per a la notificació, usant el senyal especificat al camp `sigev_signo`. Si el procés que rep el senyal obté informació extra a través d'un objecte de tipus `siginfo_t`, llavors pot obtenir informació extra:

- El camp `si_value` valdrà `sigev_value.sival_int`.
- El camp `si_code` valdrà `SI_MESGQ`.
- El camp `si_signo` valdrà `sigev_signo`.
- El camp `si_pid` valdrà el PID del procés que ha enviat el missatge.
- El camp `si_uid` valdrà el RUID (*real user ID*) del procés que ha enviat el missatge.

Al capítol 3 no s'explica el tipus `siginfo_t`; el lector interessat pot consultar el capítol 11 per a més detalls.

`SIGEV_THREAD` Es registra el procés per a la notificació. Quan arriba un senyal, es crida la funció apuntada per `sigev_notify_function` com si s'hagués cridat des d'un nou thread. Llavors, `sigev_notify_attributes` apunta a `NULL` (creació del thread amb atributs per defecte) o bé a un objecte de tipus `pthread_attr_t` amb els atributs del thread a cridar.

Els threads s'expliquen a la tercera part d'aquest llibre. Per tant, en aquest capítol només explicarem la notificació per senyal.

Exemple El següent exemple està basat en [Kerrisk, 2010, pàgines 1080–1081]. El programa rep el nom d'una cua de missatges per línia de comandes, i espera que arribi un missatge a través de la notificació per senyals.

```
1  #include <fcntl.h>
2  #include <mqueue.h>
3  #include <signal.h>
4  #include <stdio.h>
5
6  static void handler (int);
7
8  int
9  main (int argc, char *argv[])
10 {
11   if (argc < 2)
12     {
13       fprintf (stderr, "Falten arguments\n");
14       return -1;
15     }
16
17   /* Obrim la cua de missatges en mode no bloquejant */
18   mqd_t cua_id = mq_open (argv[1], O_RDONLY | O_NONBLOCK | O_CREAT);
19   if (cua_id < 0)
20     {
21       perror (argv[1]);
22       return -1;
23     }
24
25   /* Obtenim la mida m xima dels missatges
26    * i creem un buffer prou gran */
27   struct mq_attr attr;
28   mq_getattr (cua_id, &attr);
29
30   char buffer[attr.mq_msgsize];
31
32   /* Bloquegem el senyal SIGUSR1 */
33   sigset_t set;
```

```
34    sigemptyset (&set);
35    sigaddset (&set, SIGUSR1);
36    sigprocmask (SIG_BLOCK, &set, NULL);
37
38    /* Instal·lem un signal handler */
39    signal (SIGUSR1, handler);
40
41    /* Ens registrem a la cua de missatges */
42    struct sigevent sev = { 0 };
43    sev.sigev_notify = SIGEV_SIGNAL;
44    sev.sigev_signo = SIGUSR1;
45    if (mq_notify (cua_id, &sev) < 0)
46      {
47        perror ("mq_notify");
48        return -1;
49      }
50
51    /* Ens esperem que arribi el senyal */
52    sigset_t nset;
53    sigemptyset (&nset);
54
55    while (1)
56      {
57        sigsuspend (&nset);
58
59        /* Ens tornem a registrar */
60        if (mq_notify (cua_id, &sev) < 0)
61          {
62            perror ("mq_notify");
63            return -1;
64          }
65
66        ssize_t sst;
67        while ((sst = mq_receive (cua_id, buffer, attr.mq_msgsize, NULL)) >= 0)
68          printf ("Message of %ld bytes\n", sst);
69      }
70  }
71
72  static void
73  handler (int sig __attribute__ ((unused)))
74  {
75    /* No cal que faci res */
76  }
```

Alguns aspectes a considerar sobre aquest exemple:

1. Obrim la cua en mode no bloquejant perquè el bucle `while` (no el bucle infinit) finalitzarà quan ja no quedi cap més missatge per llegir, en lloc de quedar-se bloquejat executant la funció `mq_receive` a l'espera d'un nou missatge.

2. Dins del bucle infinit primer ens tornem a registrar abans de procedir a llegir els missatges. Podria ser que entre la lectura de l'últim missatge i el tornar-se a registrar arribés un nou missatge, i per tant el procés es registraria a la cua quan aquesta no estaria buida, la qual cosa provocaria que, abans de rebre una notificació, primer s'hauria de buidar la cua.

5.6.7 Algunes observacions

Comparades amb les pipes i les fifos, les cues de missatges tenen l'avantatge de poder associar un tipus o prioritat amb cada missatge. Ara bé, les cues de missatges de System V tenen els següents desavantatges:

1. No fan servir file descriptors, sinó identificadors propis. Per tant, no es poden emprar funcions com ara `select` and `poll` amb cues de missatges, amb els inconvenients que això suposa (imaginem-nos que volem multiplexar entre un socket i una cua de missatges).

2. No fan servir noms d'arxius, sinó claus, la qual cosa afegeix complexitat.

3. El sistema operatiu no porta control sobre quins processos utilitzen una cua de missatges, com sí passa amb les pipes i les fifos. Per tant, és difícil de saber, per a una aplicació, si pot eliminar una cua de missatges sense provocar pèrdua de dades, i també és difícil assegurar-se que s'eliminen les cues que ja no es fan servir.

Respecte les cues de missatges de System V, les cues POSIX presenten les següents característiques:

1. En Linux, els seus identificadors són file descriptors, i per tant es poden utilitzar les funcions `select` i `poll`.

2. Ara bé, a l'hora de llegir un missatge, sempre es llegeix per prioritat, de la més alta a la més baixa; en canvi, en System V, es pot llegir per prioritat o seleccionar un missatge d'un tipus determinat.

3. Les cues POSIX disposen del mecanisme de notificació, que no existeix en les cues de System V.

Capítol 6

Semàfors i memòria compartida

6.1 Introducció

El propòsit d'aquest capítol és explicar uns mecanismes de comunicació entre processos de molta importància: la memòria compartida i els semàfors. Cal deixar clar que aquest capítol no es proposa explicar la teoria de regions crítiques, exclusió mútua, etc; s'assumeix que el lector té uns coneixements suficients sobre aquests conceptes. Durant l'exposició del material es faran referència a aquests conceptes i problemes, però sense entrar en el seu fons. L'exposició del material començarà per la memòria compartida, ja que acostuma a ser més didàctic explicar primer el "problema" que la "solució".

Cal tenir en compte alguns aspectes concrets:

- En unix hi ha dues implementacions de semàfors i memòria compartida: els mecanismes de "System V" i els de l'estàndar posix. Els primers tenen una interfície més sofisticada (sobretot en el cas dels semàfors), però fa més temps que existeixen i per tant hi ha més implementacions de unix que els inclouen. Les versions més recents de Linux (a partir de la versió 2.6.6 del kernel) inclouen ambdues implementacions.
- La funció `mmap` ofereix un tercer mecanisme per a crear memòria compartida, del qual se'n parla a l'apartat 10.6.4.
- Els mútexs, estrictament parlant, només existeixen en la llibreria de threads. Ara bé, és trivial usar els semàfors com si fossin mútexs. Al parlar de semàfors s'aclarirà quina diferència hi ha entre ells.

Al parlar de memòria compartida, en aquest capítol es veurà únicament el mecanisme de "System V", ja que el de l'estàndar POSIX implica l'ús de la funció `mmap`, que es veurà a la segona part d'aquest llibre (apartat 10.6). En canvi, pel que fa a semàfors, s'explicaran les dues variants.

6.2 Memòria compartida

6.2.1 Introducció

Per defecte, dos processos no comparteixen cap zona de memòria (excepte aquella part dedicada a les instruccions, com podria ser el cas que dos usuaris executessin un mateix programa, com ara `ViM`), de manera que si es vol crear una zona de memòria compartida entre dos processos cal demanar-ho explícitament. Això es pot aconseguir de diferents maneres:

1. Crear el procés amb la funció `clone` i activar el flag que fa que pare i fill comparteixin l'espai d'adreces (`CLONE_VM`). La funció `clone` és un mecanisme de molt baix nivell, i per tant molt poc usat (excepte per a implementar llibreries de threads). En aquest capítol no es parlarà d'aquest mecanisme.

2. Crear threads en lloc de processos. En aquest capítol tampoc se'n parlarà.

3. Demanar explícitament una zona de memòria compartida (que serà l'única zona de memòria que compartiran). És del que es parlarà en aquest apartat.

Com a mecanisme de comunicació entre processos, la memòria compartida ofereix les següents característiques:

1. És molt més ràpida que les pipes o les fifos, i que qualsevol altre mecanisme.

   Quan un procés vol escriure en una pipe, el sistema operatiu copia les dades del buffer del procés a un buffer del sistema operatiu. Quan un altre procés vol llegir aquestes mateixes dades, es copien del buffer del sistema operatiu al buffer del procés que llegeix. A més a més, per a llegir o escriure de memòria compartida no cal cridar cap funció.

2. Necessita sincronització explícita: si dos o més processos volen escriure alhora a una mateixa zona de memòria compartida, es poden produir col·lisions. En el cas de les pipes i les fifos, per exemple, aquest problema no existeix.

3. No hi ha cap notificació que s'ha produït una escriptura.

   Quan un procés escriu a una pipe, l'altre procés ho pot "notar" perquè una crida a `read` retorna o `select` marca un file descriptor com a disponible per lectura. En canvi, no hi ha cap mena de notificació quan un procés escriu a una zona de memòria compartida.

L'ús de memòria compartida acostuma a consistir en els següents passos:

1. Es crida la funció `shmget` per crear una nova regió de memòria compartida, o obtenir l'identificador d'una zona que ja existeix (per exemple, creada per un altre procés que no té cap relació amb el nostre).

2. Es crida la funció `shmat` per tal que la regió de memòria compartida passi a formar part de l'espai d'adreces del procés.

3. A partir d'aquí, la regió de memòria compartida és com qualsevol altra zona de memòria del procés (igual que si s'hagués demanat amb `malloc`):

4. Es crida la funció `shmdt` per a "renunciar" a seguir usant la regió de memòria compartida. Aquest pas és opcional, i es fa automàticament quan un procés mor.

5. Es crida la funció `shmctl` per a eliminar la regió de memòria compartida. La regió s'elimina només quan finalitzen tots els processos que l'estaven usant (o sigui, que encara no havien cridat `shmdt`). Només cal que un procés cridi aquesta funció.

6.2.2 Crear o obrir una zona de memòria compartida

Per a crear (o obrir) una zona de memòria compartida es disposa de la funció `shmget`:

```
#include <sys/shm.h>

int shmget (key_t key, size_t size, int shmflg);
```

on `key` és una clau obtinguda amb la funció `ftok` o `IPC_PRIVATE` (en aquest segon cas, s'està creant una nova regió de memòria compartida. La mida de la zona de memòria compartida és `size` (si la zona ja existeix, `size` no pot ser superior a la mida de la zona ja creada), i `shmflags` és zero o la combinació (amb l'operador `|`) d'un o més dels valors següents:

`IPC_CREAT` En cas que no hi hagi cap regió de memòria compartida identificada per `key`, crear-ne una de nova. Si `key` és `IPC_PRIVATE`, llavors aquest flag ha d'estar present.

`IPC_EXCL` Si el flag anterior és present i existeix una regió de memòria identificada per `key`, generar un error (la funció retorna -1).

Un cop oberta, cal cridar la funció `shmat`:

```
#include <sys/shm.h>

void *shmat (int shmid, const void *shmaddr, int shmflg);
```

on `shmid` és un enter retornat per la funció `shmget`, `shmaddr` acostuma a ser `NULL`[1] i `shmflg` acostuma a ser zero (o `SHM_RDONLY` si es vol evitar fer escriptures a la zona de memòria compartida).

Els fills hereten les regions de memòria compartida del pare; les funcions de la família `exec` alliberen les zones de memòria compartida a l'executar el nou programa.

6.2.3 Eliminar una zona de memòria compartida

Quan un procés no vol seguir emprant una zona de memòria compartida, pot cridar la funció `shmdt`:

```
#include <sys/shm.h>

int shmdt (const void *shmaddr);
```

on `shmaddr` és una adreça retornada per `shmat`. La funció `shmdt` no elimina la zona de memòria, tan sols desvincula el procés de la regió. Per tal d'eliminar la zona de memòria cal cridar la funció `shmctl`:

```
#include <sys/shm.h>

int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

on `shmid` és un enter retornat per `shmget`, `cmd` val `IPC_RMID` i `buf` serà un punter `NULL`.

6.2.4 Exemple

És realment difícil donar un exemple útil de com usar la memòria compartida sense haver d'usar semàfors (o qualsevol altre mecanisme d'exclusió mútua). Per aquest motiu s'ofereix un exemple realment senzill, suficient per a il·lustrar les crides a les funcions que s'han explicat.

```
1  #include <stdint.h>
2  #include <stdio.h>
3  #include <sys/ipc.h>
4  #include <sys/shm.h>
5
6  int
7  main (void)
8  {
9    int shm =
10     shmget (IPC_PRIVATE, sizeof (uint64_t), IPC_CREAT | IPC_EXCL | 0600);
```

[1] És possible especificar una adreça, però hi ha molts inconvenients en fer-ho. Per a una explicació més detallada, es pot consultar la pàgina 1000 de Kerrisk [2010].

```
11    if (shm < 0)
12      {
13        perror ("shmget");
14        return -1;
15      }
16
17    uint64_t *valor = shmat (shm, 0, 0);
18    if (valor == NULL)
19      {
20        perror ("shmat");
21        return -1;
22      }
23
24    /* Fer coses amb la mem ria compartida */
25
26    shmdt (valor);
27    shmctl (shm, IPC_RMID, NULL);
28  }
```

6.3 Semàfors

6.3.1 Introducció

L'exposició dels semàfors s'articularà a partir de les funcionalitats que proveeixen (creació, destrucció, operacions, etc), i per a cadascuna d'elles es presentaran les dues interfícies: la del Sistema V i la de l'estàndar POSIX. Les dues operacions principals dels semàfors reben una gran varietat de noms. En aquest capítol se seguirà la nomenclatura presentada per Edsger W. Dijkstra, creador dels semàfors, i que es basa en el nom de dos verbs de l'holandès:

- L'operació *P* (del verb holandès *proberen*) significa decrementar el comptador associat al semàfor. Aquesta operació també rep el nom de *wait* o *down*. Quan el semàfor és binari (un *mutex*) també rep el nom de *lock*.
- L'operació *V* (del verb holandès *verhogen*) significa incrementar el comptador associat al semàfor. Aquesta operació també rep el nom de *signal*, *post* o *up*. Quan el semàfor és binari, també rep el nom de *unlock*.

Es recorda, un cop més, que el lector està suficientment familiaritzat amb la problemàtica associada als semàfors (és a dir, per què són necessaris). En aquest capítol es presenten els tipus de semàfors que hi ha en UNIX i com implementar les diferents operacions; no es tractaran temes de disseny com per exemple quants semàfors emprar en un cas concret, etc.

6.3.2 Creació de semàfors

6.3.2.1 Semàfors System V

La funció `semget` serveix per a crear un semàfor (o bé per obtenir un semàfor ja creat per un altre procés):

```
#include <sys/sem.h>

int semget (key_t key, int nsems, int semflg);
```

on `key` identifica un semàfor que ja existeix (o bé `IPC_PRIVATE` per a un nou semàfor), `nsems` és el número de semàfors a crear i `flags` acostuma a valdre `IPC_CREAT` (si es crea un nou semàfor) i els permisos del semàfor. Per exemple:

```
int semid;
semid = semget (IPC_PRIVATE, 1, IPC_CREAT | S_IRUSR | S_IWUSR);
```

crea un nou semàfor amb permisos de lectura i escriptura per a l'usuari.

Convé fer un comentari addicional sobre l'argument `nsems`. Els semàfors de System V permeten operar amb conjunts de semàfors, i fer operacions que afectin el conjunt. En aquest capítol es consideraran sempre conjunts d'*un sol* semàfor.

Els semàfors de System V són heretats pels fills de forma automàtica.

Inicialització Un cop el semàfor està creat, encara no està a punt per a fer-lo servir: cal donar-li un valor inicial, és a dir, inicialitzar-lo. No hi ha una funció específica a tal fi, sinó que cal cridar la funció `semctl`, que permet realitzar una varietat de funcionalitats:

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

int semctl (int semid, int semnum, int cmd,
            ... /* union semun arg */);
```

Per més estrany que pugui semblar, l'usuari ha de definir el tipus `union semun`, de la següent manera:

```
union semun
{
  int val;
  struct semid_ds *buf;
  unsigned short *array;
  struct seminfo *__buf;
};
```

En el cas de la inicialització d'un semàfor, cal passar els següents arguments:

- L'argument `semid` correspon a l'identificador obtingut amb la funció `semget`.
- Pel que fa a l'argument `semnum`, el número (el primer és el 0) del semàfor a inicialitzar. Cal recordar que la interfície de System V permet que, per a un sol identificador `semid`, hi hagi més d'un semàfor. En aquest llibre no considerarem aquest cas; per tant, `semnum` valdrà 1.
- L'argument `cmd` valdrà `SETVAL`.
- El camp `val` de l'argument `arg` contindrà el valor inicial amb el qual cal inicialitzar el semàfor.

El següent exemple mostra com crear i inicialitzar un semàfor:

```
1  #include <stdio.h>
2  #include <sys/ipc.h>
3  #include <sys/sem.h>
4  #include <sys/stat.h>
5  #include <sys/types.h>
6
7  union semun
8  {
9    int val;
10   struct semid_ds *buf;
11   unsigned short *array;
12   struct seminfo *__buf;
13 };
14
15 int
16 main (void)
17 {
18   int semid = semget (IPC_PRIVATE, 1, IPC_CREAT | S_IRUSR | S_IWUSR);
19   if (semid < 0)
20     {
21       perror ("semget");
22       return -1;
23     }
24
25   union semun s;
26   s.val = 1;
27
28   if (semctl (semid, 0, SETVAL, s) < 0)
29     {
30       perror ("semctl");
31       return -1;
32     }
33
```

```
34    /* Fer coses amb el sem for */
35
36    semctl (semid, 0, IPC_RMID);
37  }
```

6.3.2.2 Semàfors POSIX

En POSIX hi ha dues menes de semàfors: semàfors amb nom i semàfors sense nom. Els primers són útils perquè els puguin emprar processos que entre ells no tinguin cap mena de relació.

Semàfors amb nom Pel que fa als semàfors amb nom, es creen i s'inicialitzen de la següent manera:

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

sem_t *sem_open (const char *name, int oflag, ...
                 /* mode_t mode, unsigned int value */);
```

El paràmetre `name` serveix per donar nom al semàfor. En Linux, un nom consisteix en una barra inclinada (/) i una cadena de com a molt 251 caràcters (que no poden ser la barra inclinada). El paràmetre `oflag` és molt similar al de la funció `open`. Per exemple, per a crear un nou semàfor especificarem `O_CREAT`, i si volem que el semàfor a crear no existeixi prèviament, especificarem també `O_EXCL`. El paràmetre `mode` indica els permisos del semàfor, com al crear un arxiu, i el paràmetre `value` indica el valor inicial del semàfor. Per exemple:

```
sem_t *mysem = sem_open ("/mysem", O_CREAT | O_EXCL, 0644, 1);
```

Semàfors sense nom Es creen i s'inicialitzen amb la funció següent:

```
#include <semaphore.h>

int sem_init (sem_t *sem, int pshared, unsigned int value);
```

Mentre que la funció `sem_open` retorna un punter a l'objecte semàfor, la funció `sem_init` requereix que se li passi un punter a un objecte prèviament creat. El paràmetre `pshared` val zero si el semàfor el comparteixen threads, i val 1 si el comparteixen processos. Cal tenir en compte, i això és molt important, que llavors l'objecte ha de residir en una zona de memòria compartida que comparteixin els processos que també compartiran el semàfor. Finalment, el paràmetre `value` indica el valor amb què s'inicialitzarà el semàfor.

En el següent exemple, es fa servir la funció `mmap` per a demanar memòria compartida. És igual el mètode que s'empri, sempre i quan sigui memòria compartida.

```
#include <semaphore.h>
#include <stdio.h>
#include <sys/mman.h>

int
main (void)
{
  sem_t *sem = mmap (NULL, sizeof (sem_t), PROT_READ | PROT_WRITE,
                     MAP_ANON | MAP_SHARED, -1, 0);
  if (sem == MAP_FAILED)
    {
      perror ("mmap");
      return -1;
    }

  if (sem_init (sem, 1, 1) < 0)
    {
      perror ("sem_init");
      return -1;
    }

  /* No cal alliberar ni la mem ria compartida
   * ni el sem for al finalitzar el proc s */
}
```

S'indica que el semàfor serà compartit entre processos i s'inicialitza a 1. La funció `sem_init` retorna -1 si hi ha hagut algun error, i zero en cas contrari.

6.3.3 Operacions amb semàfors

6.3.3.1 System V

L'API de semàfors de System V és força complicada, com s'anirà veient. No hi ha una funció específica per a cadascuna de les operacions, sinó que la funció `semop` serveix per a totes:

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

int semop (int semid, struct sembuf *sops, unsigned int nsops);
```

on `semid` és l'identificador retornat per `semget`, `sops` apunta a un element (però podria ser un array) de tipus `struct sembuf` i `nsops` indica a quants elements apunta `sops` (típicament 1). El tipus `struct sembuf` està definit de la següent manera:

```
struct sembuf
{
```

```
  unsigned short sem_num;
  short sem_op;
  short sem_flg;
};
```

- `sem_num` indica el número de semàfor sobre el qual s'efectuarà l'operació. Si, com és el cas més habitual, l'identificador `semid` conté un sol semàfor, `sem_num` valdrà zero.

- `sem_flg` és una màscara de bits que permet especificar cap, un, o els dos flags següents:

  `IPC_NOWAIT` Evita que `semop` bloquegi un procés. En lloc de bloquejar el procés, `semop` retorna -1 amb el codi d'error `EAGAIN`.

  `IPC_UNDO` Desfà les operacions fetes en un semàfor quan un procés mor. És útil en el cas que un procés mor mentre té bloquejat un semàfor, de manera que la resta de processos no queden bloquejats indefinidament. Per a una explicació més detallada d'aquest flag es recomana consultar [Kerrisk, 2010, apartat 47.8, pàgines 986–988].

- `sem_op` representa l'operació a realitzar:

  - Si `sem_op` és superior a zero, el seu valor s'afegeix al valor del semàfor, de manera que algun dels processos bloquejats en el semàfor (esperant poder decrementar el seu valor) queden desbloquejats. El procés cal que tingui permís d'escriptura sobre el semàfor. Vindria a ser l'operació *P*.
  - Si `sem_op` és zero, es comprova si el valor del semàfor és zero. En cas afirmatiu, `semop` retorna immediatament; en cas contrari, el procés queda bloquejat fins que el valor del semàfor val zero. El procés ha de tenir permís de lectura sobre el semàfor.
  - Si `sem_op` és inferior a zero, aquest valor es resta al valor del semàfor (és a dir, el valor del semàfor es decrementa). Si el valor actual del semàfor és igual o superior al valor absolut de `sem_op`, `semop` retorna immediatament; en cas contrari, el procés queda bloquejat fins que el semàfor pren un valor que permet realitzar l'operació sense que el semàfor prengui un valor negatiu. El procés ha de tenir permís d'escriptura sobre el semàfor. Vindria a ser l'operació *V*.

  Cal tenir en compte que `semop` pot retornar si el procés rep un senyal o si es destrueix el semàfor.

Exemple

6.3.3.2 POSIX

L'API de semàfors de POSIX és força entenedora.

Operació *P* L'operació *P* (també anomenada *lock*, *wait* o *down*) es realitza amb la funció següent:

```
#include <semaphore.h>

int sem_wait (sem_t *sem);
```

on `sem` apunta a un semàfor prèviament creat (ja sigui amb nom o sense nom). Si el valor del semàfor és superior a zero, la funció retorna immediatament (havent decrementat el valor del semàfor). Si és zero, la funció bloqueja el procés (o thread) fins que el valor del semàfor torna a ser superior a zero; en aquest cas, decrementa el valor i la funció retorna.

Si un procés bloquejat per `sem_wait` rep un senyal, llavors `sem_wait` retorna immediatament -1 amb el codi d'error `EINTR` (a la variable global `errno`), independentment de si a l'establir el signal handler es va activar el flag `SA_RESTART` (vegeu els capítols de senyals per a més detalls).[2]

A part de la funció `sem_wait` hi ha una altra funció, anomenada `sem_trywait` (que rep els mateixos arguments), que té un funcionament idèntic al de `sem_wait` però amb una diferència: si el valor del semàfor és zero, aquesta funció no bloqueja el procés sinó que retorna -1 amb el codi d'error `EAGAIN`.

Hi ha una altra funció, anomenada `sem_timedwait`, que permet establir un timeout màxim d'espera, de manera que el procés no quedi mai bloquejat indefinidament.

Operació *V* L'operació *V* (també anomenada *unlock*, *signal*, *post* o *up*) es realitza amb la funció següent:

```
#include <semaphore.h>

int sem_post (sem_t *sem);
```

on `sem` apunta a un semàfor prèviament creat (ja sigui amb nom o sense nom). Si hi ha un o més processos (o threads) bloquejats en aquest semàfor, llavors un d'ells es desbloquejarà. El programador no pot saber ni determinar quin d'aquests processos serà el que es desbloquegi.

[2] En d'altres implementacions de UNIX aquest flag sí fa que la funció no s'interrompi.

Consultar el valor La funció `sem_getvalue` permet consutar el valor que un semàfor té en un determinat moment:

```
#include <semaphore.h>

int sem_getvalue (sem_t *sem, int *sval);
```

El valor del semàfor s'ubicarà a la memòria apuntada per `sval`. Cal tenir en compte dos aspectes:

1. En Linux, el valor d'un semàfor mai baixa de zero (és a dir, mai serà negatiu).
2. Quan la funció `get_semvalue` retorna, pot ser que el valor del semàfor hagi canviat, és a dir, que la informació que dóna la funció estigui antiquada.

6.3.4 Eliminació de semàfors

6.3.4.1 System V

No hi ha una funció específica per eliminar un semàfor, sinó que s'empra la funció `semctl`, que ja s'ha vist anteriorment. Convé tenir en compte que els semàfors de System V cal eliminar-los explícitament. Si un procés no elimina els semàfors que usa, el sistema operatiu no ho fa per ell, la qual cosa obliga l'usuari o l'administrador a eliminar-los manualment (amb les comandes `ipcs` i `ipcrm`).

Per a l'eliminació d'un semàfor, cal passar els següents arguments a la funció `semctl`:

- L'argument `semid` correspon a l'identificador obtingut amb la funció `semget`.
- L'argument `semnum` s'ignora.
- L'argument `cmd` ha de valdre `IPC_RMID`.
- No es requereix passar cap més argument (és a dir, no cal passar l'argument `arg`).

6.3.4.2 POSIX

Semàfors amb nom Quan un procés vol deixar d'emprar un semàfor amb nom, disposa de la funció següent:

```
#include <semaphore.h>

int sem_close (sem_t * sem);
```

Executar un nou programa (amb `execve` o similar) o finalitzar l'execució del procés tanca automàticament el semàfor. Ara bé, per a destruir-lo (que només es fa fins que no queda cap procés usant-lo) hi ha la següent funció:

```
int sem_unlink (const char *name);
```

Semàfors sense nom Per als semàfors sense nom hi ha la funció següent:

```
#include <semaphore.h>

int sem_destroy (sem_t *sem);
```

Un semàfor s'ha de destruir només quan no hi ha cap thread o procés bloquejat esperant el semàfor, i s'ha de fer sempre abans que la memòria apuntada per `sem` deixi d'existir. Per exemple, si `sem` apunta a una variable automàtica, cal cridar `sem_destroy` abans que aquesta variable surti d'àmbit; si es tracta de memòria compartida, abans d'alliberar la memòria compartida.

6.3.5 Comparació entre semàfors POSIX i System V

En aquest capítol s'han vist dues implementacions de semàfors: la de POSIX i la de System V. En aquest apartat es faran alguns apunts comparan les dues interfícies:

- La interfície POSIX és molt més senzilla que la de System V, sense pèrdua de funcionalitat.
- La interfície de System V permet agrupar diferents semàfors sota un sol identificador, tot i que pràcticament no s'empra mai.
- Els semàfors POSIX creen i inicialitzen el semàfor alhora, sense necessitar dues funcions diferents.
- Quan hi ha molta contenció sobre el semàfor, el rendiment de les dues interfícies és similar. Ara bé, quan no hi ha contenció (és a dir, quan la majoria d'operacions *P* no bloquegen el procés), el rendiment dels semàfors POSIX és considerablement millor.[3]
- La interfície de System V és anterior a la de POSIX i més portable, és a dir, disponible en més implementacions de UNIX. En Linux mateix, els semàfors POSIX estan disponibles només a partir del kernel 2.6.
- Els semàfors POSIX no tenen equivalent de `IPC_UNDO`.
- Quan es vol compartir un semàfor POSIX entre diferents processos, cal ubicar-los en una zona de memòria compartida.
- Els semàfors POSIX s'alliberen automàticament al finalitzar els processos involucrats, cosa que no passa amb els de System V, que cal destruir-los explícitament.

[3] Els semàfors de System V requereixen una crida al sistema qualsevol que sigui l'operació que realitzen; en canvi, els semàfors POSIX només requereixen la crida al sistema quan cal arbitrar entre dos processos que volen bloquejar el semàfor.

Capítol 7

Sockets: conceptes fonamentals

7.1 Introducció

De tots els mecanismes de comunicació entre processos de què disposa UNIX, els sockets són els únics que permeten comunicar processos que no s'executen a la mateixa màquina. Emprats d'aquesta manera, la comunicació es fa mitjançant protocols com ara IP i TCP, dissenyats específicament per a aquesta finalitat. No cal dir, doncs, que els sockets són el mecanisme que permeten la transmissió de dades a través d'internet.

Aquest capítol assumeix, per una banda, que el lector coneix el sistema d'entrada i sortida de UNIX; per altra banda, que el lector té un coneixement suficient dels protocols d'internet, específicament IP i TCP. Cas de no ser així, és important que el lector repassi aquests temes, en cas contrari s'exposa a no entendre l'exposició dels sockets que es farà en aquest capítol.

El material que es presenta en aquest capítol se centra als sockets que empren el protocol TCP, ja que aquest prococol és, de llarg, el més usat a Internet (juntament amb UDP), i que el s'empra típicament a les pràctiques de l'assignatura.

S'exhorta els alumnes a llegir completament aquest capítol abans d'escriure una sola línia de codi relacionada amb sockets. Molts errors es poden evitar si es coneixen i es comprenen els detalls que afecten la programació amb sockets.

7.2 Conceptes

Un socket és un file descriptor que permet la transmissió de dades a través d'una connexió TCP .Hi ha sockets per a transmetre dades a través d'altres protocols com ara UDP, SCTP o DCCP, a través del protocol IP directament (per exemple per enviar paquets ICMP), a través d'un protocol d'enllaç com ara Ethernet (molt útils per a fer *sniffers*), i fins i tot sockets que permeten modificar la taula d'enrutament; en aquest capítol només es veuran els sockets per a comunicacions TCP.

Un socket és *actiu* si serveix per a iniciar connexions; es diu que és *passiu* si serveix per a rebre connexions. D'aquesta manera, un client emprarà un socket actiu per a connectar-se un servidor, que haurà obert un socket passiu per a rebre noves connexions. Aquests conceptes s'aniran aclarint al llarg del capítol, com així els mecanismes per a iniciar, rebre i acceptar connexions.

El primer pas "universal" per a treballar amb sockets és crear un file descriptor amb la funció següent:

```
#include <sys/socket.h>

int socket (int family, int type, int protocol);
```

La combinació dels valors `family`, `type` i `protocol` determinaran el tipus de socket i els protocols a usar. Per a TCP, aquests valors són:

- El paràmetre `family` serà `AF_INET` (per a IPv4) o `AF_INET6` (per a IPv6).
- El paràmetre `type` serà `SOCK_STREAM`.
- El paràmetre `protocol` valdrà zero, o bé la "constant" `IPPROTO_TCP`, definida a `netinet/in.h`

Si s'ha pogut crear el socket, la funció `socket` retorna un enter positiu; si no s'ha pogut crear retorna -1. Si posteriorment no s'indica el contrari, els sockets són actius.

Exemple

```
1  #include <netinet/in.h>
2  #include <stdio.h>
3  #include <string.h>
4  #include <sys/socket.h>
5
6  int
7  main (void)
8  {
9    int sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
10   if (sockfd < 0)
11     {
```

```
12        perror ("socket");
13        return -1;
14      }
15  }
```

7.3 Servidor: obertura passiva

Abans que un client pugui iniciar una connexió, algun servidor s'ha d'haver preparat per a rebre connexions. A tal fi farà dues coses:

1. Un cop creat el socket, li assignarà un port (i, si vol, una adreça IP, tot i que típicament serà només un port).
2. Convertirà el socket en passiu.

En tot aquest capítol s'assumiran sockets bloquejants, que és el comportament per defecte. Quan els sockets operen en mode no bloquejant, el comportament d'algunes funcions canvia — particularment la funció `connect` i totes les que transmeten dades.

7.3.1 Funció *bind*

Per al primer pas, es disposa d'un parell d'estructures de dades i d'una funció. Les estructures de dades són les següents:

```
#include <netinet/in.h>

typedef uint32_t in_addr_t;

struct in_addr
{
  in_addr_t s_addr;
};

struct sockaddr_in
{
  uint8_t sin_len;
  sa_family_t sin_family;
  in_port_t sin_port;
  struct in_addr sin_addr;
  char sin_zero[8];
};
```

El primer *struct* defineix què és una adreça IP: un enter sense signe de 32 bits, **en network byte order**, o sigui, **big endian**. El segon *struct* defineix l'estructura que permet establir l'adreça IP i el port TCP:

- El camp `sin_len` indica la mida, en bytes, de tot l'objecte. El programador no s'ha de preocupar en absolut d'aquest camp.
- El camp `sin_family` indica la versió d'IP que es fa servir; en aquesta mena de socket ha de valdre `AF_INET`. **Nota**: si es vol emprar IPV6, cal emprar unes altres estructures, concretament `struct sockaddr_in6`. Aquest capítol se centrarà exclusivament en IPV4.
- El camp `sin_port` és un enter de 16 bits (en **network byte order**, o sigui, **big endian**) que indica el port TCP.
- El camp `sin_addr` conté l'adreça IP. No és un enter directament, sinó una estructura que conté l'enter.
- El camp `sin_zero` és de padding. El programador no s'ha de preocupar en absolut d'aquest camp.

El fet que aquesta estructura sigui més complexa del que aparentment seria necessari és que les funcions que es veuran en aquest capítol es poden aplicar, més o menys, a molts tipus de sockets, els quals tenen estructures diferents. Per això hi ha el camp `sin_family`, perquè la funció pugui saber de quin tipus de socket es tracta.

La funció `bind` serveix per a vincular una adreça especificada en un objecte de tipus `struct sockaddr_in` a un determinat socket:

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

int bind (int sockfd,
          const struct sockaddr *myaddr,
          socklen_t addrlen);
```

Convé notar que la funció no rep un punter a `struct sockaddr_in`, sinó a una estructura genèrica de tipus `struct sockaddr`. El problema (des del punt de vista de llenguatge) es resol fàcilment amb un *cast*. El programador té tres opcions:

- No especificar ni adreça IP ni port TCP. En aquest cas, el camp `sin_port` valdrà zero, i el camp `sin_addr.s_addr` valdrà `INADDR_ANY`, que de fet és zero. D'aquesta manera, serà el sistema operatiu el que decidirà tant l'adreça com el port.
- Especificar el port o l'adreça IP. El programador tria quin dels dos paràmetres vol especificant un valor concret a `sin_port` o `sin_addr`, deixant l'altre "a zero". El cas més comú, majoritari, per a un servidor, és especificar únicament el port.
- Especificar el port i l'adreça IP. El programador especifica valor per a `sin_port` i `sin_addr`.

Per a donar valor a `sin_port` cal tenir en compte que el número de port ha de ser en *network byte order*, i per a això hi ha les dues funcions següents:

```
#include <arpa/inet.h>

uint16_t htons (uint16_t s);
uint16_t ntohs (uint16_t s);
```

- La funció `htons` (*host to network short*) converteix l'enter de 16 bits `s` al format de la xarxa, és a dir, a *big endian*. Si el host ja treballa en *big endian*, la funció no altera el valor. Les màquines Intel treballen en *little endian*, d'altres arquitectures no.
- La funció `ntohs` (*network to host short*) converteix l'enter de 16 bits `s` al format del host, ja sigui *big endian* o *little endian*.

Per a donar valor a `sin_addr`, hi ha dues funcions que permeten convertir una adreça en format ASCII a format binari (enter de 32 bits) i vice-versa:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int inet_aton (const char *cp, struct in_addr *inp);
char *inet_ntoa (struct in_addr in);
```

- La funció `inet_aton` rep una cadena de text amb una adreça `ip` i la converteix a format binari; el resultat el guarda a l'objecte apuntat per `inp`.
- La funció `inet_ntoa` rep una adreça en format binari, i retorna un punter a una cadena que conté la representació ASCII de l'adreça. Cal tenir en compte que el valor de retorn és l'adreça d'una zona de memòria estàtica, i per tant no es pot alliberar i se sobreescriu amb cada crida a `inet_ntoa`.

La funció `bind` intentarà assignar, al socket, l'adreça i el port especificat. Pel que fa al port, pot fallar per tres motius principals:

1. Perquè ja hi ha un altre procés que té un socket amb el mateix port. Per exemple, si hi ha un servidor web que escolta el port 80, no es pot obrir cap altre procés que també vulgui escoltar el port 80.
2. Perquè hi ha una connexió oberta cap al port que es vol assignar al socket. Per exemple, si un procés obre (assigna al socket) un port, rep una connexió i fa un procés fill que l'atengui, i després el pare es mor (però no el fill), llavors si es torna a executar el programa, aquest no podrà tornar a obrir el port.

3. Perquè hi ha una connexió en l'estat `TIME_WAIT` que empra el port que es vol obrir. Per a una explicació d'aquest estat, el lector pot consultar [Stevens, 1994, pàgines 242-245].

Per al primer cas, no hi ha res a fer. Per als dos següents, hi ha una opció de socket anomenada `SO_REUSEADDR` que permet assignar el port en aquestes circumstàncies. Per a les opcions de sockets, consulteu l'apartat 12.3.

Cal tenir en compte, a més a més, que els ports 1–1023 estan reservats i només els pot obrir l'usuari `root`.

Exemple El programa següent crea un socket i li assigna un port, que es rep per línia de comandes. Cal tenir en compte que no es comprova la validesa del port en qüestió. L'exemple també conté una crida a la funció `listen`, que s'explica a l'apartat següent.

```
1  #include <arpa/inet.h>
2  #include <netinet/in.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <sys/socket.h>
7
8  int
9  main (int argc, char *argv[])
10 {
11   if (argc < 2)
12     {
13       printf ("Falta indicar el port\n");
14       return -1;
15     }
16
17   int sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
18   if (sockfd < 0)
19     {
20       perror ("socket");
21       return -1;
22     }
23
24   struct sockaddr_in s_addr;
25   memset (&s_addr, 0, sizeof (s_addr));
26   s_addr.sin_family = AF_INET;
27   s_addr.sin_port = htons (atoi (argv[1]));
28   s_addr.sin_addr.s_addr = INADDR_ANY;
29
30   if (bind (sockfd, (void *) &s_addr, sizeof (s_addr)) < 0)
31     {
32       perror ("bind");
33       return -1;
```



```
34      }
35
36    listen (sockfd, 3);
37  }
```

7.3.2 Funció *listen*

La funció `bind` no converteix el socket en passiu, és a dir, no el prepara per a rebre connexions. De fet, la funció `bind` fins i tot la pot fer servir un client. Per tal de convertir el socket en passiu i poder rebre connexions, cal cridar la funció `listen`:

```
#include <sys/socket.h>

int listen (int sockfd, int backlog);
```

La finalitat d'aquesta funció és doble:

1. Indica que per un determinat socket `sockfd` es podran rebre connexions TCP.
2. Indica quantes connexions pendents hi pot haver per aquest socket. Es considera que una connexió està pendent quan s'ha rebut el primer segment SYN del *three-way handshake*, però encara no s'ha completat el *handshake*; o bé que el *handshake* s'ha completat però l'aplicació encara no s'ha donat per assabentada de la nova connexió.

   Més específicament: per a un socket que accepta connexions, el sistema operatiu manté dues cues de peticions de connexió: la primera consisteix en connexions incompletes, és a dir, aquelles en què s'ha iniciat (però no completat) el *three-way handshake*; la segona consisteix en connexions completes, però per les quals l'aplicació ha de cridar la funció `accept`.

   La suma de les dues cues no pot excedir `backlog` peticions.

Un cop arribats en aquest punt, el servidor ja està a punt per a rebre connexions.

7.4 Client: obertura activa

Quan un client vol iniciar una connexió contra un servidor, farà dues coses:

1. Un cop creat el socket, pot opcionalment especificar l'adreça i/o el port origen de la connexió. Per a la majoria de clients, no és en absolut necessari.
2. Especificarà contra quina adreça i port vol iniciar la connexió.

El primer pas es fa exactament igual que en el cas del servidor, és a dir, amb la funció `bind`. El segon pas es fa amb la funció `connect`.

7.4.1 Funció *connect*

La funció `connect` s'encarrega de dur a terme el *three-way handshake* del protocol TCP.

```
#include <sys/socket.h>

int connect (int sockfd,
             const struct sockaddr *servaddr,
             socklen_t addrlen);
```

`sockfd` és el file descriptor del socket, `servaddr` apunta a l'adreça del servidor i `addrlen` és la mida, en bytes, de l'objecte apuntat per `servaddr`. Per defecte, la funció `connect` no retorna fins que passa algun dels tres esdeveniments següents:

1. S'ha completat el *three-way handshake* amb èxit. En aquest cas, `connect` retorna zero.
2. La connexió ha sigut rebutjada pel servidor (amb un segment de reset). En aquest cas, `connect` retorna -1 amb un error de "Connection refused" o similar.
3. El servidor no ha respost al segment SYN (o les respostes s'han perdut) i s'han esgotat les retransmissions. En aquest cas, `connect` retorna -1 amb un error de "Connection timed out" o similar.

Quan `connect` falla, el codi d'error es troba a la variable global `errno`, declarada a `errno.h`, i es pot emprar la funció `perror` o `strerror` per mostrar-ne el missatge (consulteu les pàgines manual per a més informació sobre aquestes dues funcions).

Exemple L'exemple següent presenta un programa que rep dos arguments per línia de comandes: una adreça IP i un port TCP. El programa crea un socket i intenta connectar-se a l'adreça i el port especificats.

```
1  #include <arpa/inet.h>
2  #include <netinet/in.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <sys/socket.h>
7
8  int
9  main (int argc, char *argv[])
10 {
11   if (argc < 3)
12     {
13       printf ("Falten par metres (IP i port dest )\n");
```

```
14        return -1;
15      }
16
17    int sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
18    if (sockfd < 0)
19      {
20        perror ("socket");
21        return -1;
22      }
23
24    struct sockaddr_in s_addr;
25    memset (&s_addr, 0, sizeof (s_addr));
26    s_addr.sin_family = AF_INET;
27    s_addr.sin_port = htons (atoi (argv[2]));
28    if (inet_aton (argv[1], &s_addr.sin_addr) == 0)
29      {
30        printf ("%s no  s una adre a IP v lida\n", argv[1]);
31        return -1;
32      }
33
34    if (connect (sockfd, (void *) &s_addr, sizeof (s_addr)) < 0)
35      {
36        perror ("connect");
37        return -1;
38      }
39  }
```

7.4.2 Funció *gethostbyname*

Al parlar de la funció `bind` (apartat 7.3.1) s'ha comentat la funció `inet_aton`, però quan es tracta de connectar-se a un servidor les adreces IP no són gaire còmodes: la gent recorda noms com ara *vela.salle.url.edu* i no 84.88.233.221; per tant, hi ha d'haver algun mecanisme per a obtenir l'adreça IP a partir d'un nom de host. D'això se n'encarrega la funció següent:

```
#include <netdb.h>

struct hostent *gethostbyname (const char *name);
```

on `name` és un nom com *vela.salle.url.edu*. La funció retorna un punter a una zona de memòria estàtica (per tant no es pot alliberar i se sobreescriu cada cop que es crida la funció) que conté informació sobre el host, incloent la seva adreça (o adreces) IP en format binari.

Exemple 1 Per a explicar com fer servir aquesta funció, s'ofereix un exemple que rep un nom d'un host (o més d'un nom) per línia de comandes, i en mostra la seva informació.

```
#include <arpa/inet.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>

int
main (int argc, char *argv[])
{
  char *ptr, **pptr;
  struct hostent *hptr;

  while (--argc > 0)
    {
      ptr = *++argv;
      if ((hptr = gethostbyname (ptr)) == NULL)
        {
          fprintf (stderr, "%s: %s\n", ptr, hstrerror (h_errno));
          continue;
        }
      printf ("Official hostname: %s\n", hptr->h_name);

      for (pptr = hptr->h_aliases; *pptr != NULL; pptr++)
        printf ("\tAlias: %s\n", *pptr);

      struct in_addr *addr;
      switch (hptr->h_addrtype)
        {
        case AF_INET:
          for (pptr = hptr->h_addr_list; *pptr != NULL; pptr++)
            {
              addr = (void *) *pptr;
              printf ("\tAddress: %s\n", inet_ntoa (*addr));
            }
          break;

        default:
          printf ("Unknown address type\n");
          break;
        }
    }

  return EXIT_SUCCESS;
}
```


Exemple 2 Addicionalment, s'ofereix l'exemple de la funció `connect` modificat per a rebre un nom de host en lloc d'una adreça IP:

```
#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>

int
main (int argc, char *argv[])
{
  if (argc < 3)
    {
      printf ("Falten par metres (host i port dest )\n");
      return -1;
    }

  int sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (sockfd < 0)
    {
      perror ("socket");
      return -1;
    }

  struct sockaddr_in s_addr;
  memset (&s_addr, 0, sizeof (s_addr));
  s_addr.sin_family = AF_INET;
  s_addr.sin_port = htons (atoi (argv[2]));

  struct hostent *host = gethostbyname (argv[1]);
  if (host == NULL)
    {
      perror ("gethostbyname");
      return -1;
    }
  bcopy (host->h_addr, &s_addr.sin_addr.s_addr, host->h_length);

  if (connect (sockfd, (void *) &s_addr, sizeof (s_addr)) < 0)
    {
      perror ("connect");
      return -1;
    }
}
```

7.5 Servidor: rebre la connexió

A l'apartat 7.3 s'ha vist com un servidor obre un socket per a rebre connexions. Amb la funció `listen` el socket passa a ser passiu, però aquesta funció no és la que rep les noves connexions. A tal fi hi ha la funció `accept`, que bloqueja el procés (excepte per a sockets no bloquejants, que no és el cas d'aquest capítol) fins que arriba una nova connexió:

```
#include <sys/socket.h>

int accept (int sockfd,
            struct sockaddr *cliaddr,
            socklen_t *addrlen);
```

on `sockfd` és un file descriptor d'un socket passiu i `cliaddr` és un punter a l'adreça de qui ha iniciat la connexió. Quan es crida la funció, `addrlen` ha d'apuntar a un enter que contingui la mida, en bytes, de la zona de memòria apuntada per `cliaddr`; un cop la funció retorna, apunta a la mida real que ocupa l'objecte apuntat per `cliaddr`. Si no es desitja saber l'adreça de qui ha iniciat la connexió, es poden passar punters `NULL` a `cliaddr` i a `addrlen`.

És molt important tenir clar que `accept` retorna **un nou file descriptor**, que és el que s'ha de fer servir per a comunicar-se amb el client. El file descriptor `sockfd` només serveix per a rebre noves connexions.

Exemple S'ha ampliat l'exemple de la funció `bind` per tal d'il·lustrar com cridar la funció `accept`:

```
1  #include <arpa/inet.h>
2  #include <netinet/in.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <sys/socket.h>
7
8  static int do_tcp_passive_open (const char *port);
9
10 int
11 main (int argc, char *argv[])
12 {
13   if (argc < 2)
14     {
15       printf ("Falta indicar el port\n");
16       return -1;
17     }
18
19   int sockfd = do_tcp_passive_open (argv[1]);
20
```

```
21    struct sockaddr_in s_addr;
22    socklen_t len = sizeof (s_addr);
23
24    int newsock = accept (sockfd, (void *) &s_addr, &len);
25    if (newsock < 0)
26      {
27        perror ("accept");
28        return -1;
29      }
30
31    printf ("Nova connexi  procedent de %s:%hu\n", inet_ntoa (s_addr.sin_addr),
32            ntohs (s_addr.sin_port));
33  }
34
35  static int
36  do_tcp_passive_open (const char *port)
37  {
38    int sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
39    if (sockfd < 0)
40      {
41        perror ("socket");
42        exit (-1);
43      }
44
45    struct sockaddr_in s_addr;
46    memset (&s_addr, 0, sizeof (s_addr));
47    s_addr.sin_family = AF_INET;
48    s_addr.sin_port = htons (atoi (port));
49    s_addr.sin_addr.s_addr = INADDR_ANY;
50
51    if (bind (sockfd, (void *) &s_addr, sizeof (s_addr)) < 0)
52      {
53        perror ("bind");
54        exit (-1);
55      }
56
57    listen (sockfd, 3);
58    return sockfd;
59  }
```

7.6 Tancament de la connexió

La funció `close` serveix, també, per a sockets, i el seu comportament depèn de si se li passa un socket actiu o passiu:

- Si es tracta d'un socket passiu, llavors s'està indicant que no es desitja rebre més connexions. Les connexions ja existents no es tanquen, però no se n'admetran de noves.

- Si es tracta d'un socket actiu, llavors s'està indicant que es desitja tancar la connexió (el que en TCP es coneix com a *active close*).

Si un file descriptor de socket està compartit entre diversos processos, no es duu a terme el tancament fins que tots ells han tancat el file descriptor.

El protocol TCP permet el que es coneix amb el nom de *half-close*, és a dir, quan una de les bandes de la connexió indica a l'altra que no té més dades a enviar, però està disposada a seguir-ne rebent. Per a un exemple de quan pot ser útil servir-se d'aquesta característica de TCP es pot consultar [Stevens, 1994, apartat 18.5, pàgines 238–240]. La funció `shutdown` permet dur a terme aquest *half-close*.

Per defecte, al tancar-se una connexió s'intenta que no es perdin dades que estaven viatjant cap al seu destinatari. Hi ha una opció de socket, anomenada `SO_LINGER`, que permet modificar aquest comportament.

7.7 Transmissió de dades

Per a enviar i rebre dades per un socket, hi ha tres opcions (que es poden simultanejar):

1. Emprar les funcions `read` i `write`, com en els arxius, pipes, etc.
2. Emprar les funcions `readv` i `writev`, com en els arxius, pipes, etc.
3. Emprar funcions específiques per a sockets, com ara `send` i `recv`, exclusives per a sockets.[1]

És molt important tenir en compte alguns aspectes del funcionament del protocol TCP per tal d'entendre com operen aquestes funcions.

El protocol TCP considera les dades enviades i rebudes com un simple flux de bytes, sense tenir en compte cap mena de *frontera* entre missatges. Quan un procés envia dades a enviar a través, per exemple, de la funció `write`, el sistema operatiu copia aquestes dades al buffer del socket (la mida del qual, dins d'uns marges, es pot modificar) i després el protocol TCP decideix com fragmenta les dades en diferents segments TCP. El fet de cridar la funció `write` (o una altra funció equivalent) no significa que, en aquell mateix instant, surti un o més segments TCP cap al destinatari.

Per a un socket bloquejant (que és com són els sockets per defecte), la funció `write` (i equivalents) bloquegen el procés fins que han aconseguit escriure tots els bytes al buffer del sistema operatiu.

Els problemes greus apareixen al llegir les dades. La funció `read` bloqueja el procés fins que apareixen dades al buffer del sistema operatiu (tenint en compte els números de seqüència del protocol TCP, és a dir, que no n'hi ha prou que arribin dades, sinó que les dades han de ser les que anaven a continuació de les últimes que

[1] Funcions com ara *sendto*, *sendmsg*, *recvfrom* i *recvmsg* no tenen massa sentit per a TCP, tot i que també es poden emprar.

es van llegir). En aquest moment `read` ja pot retornar. Ara bé, quantes dades haurà llegit? No necessàriament totes les que hi havia, o totes les que s'han demanat (sí podem estar segurs que no en llegirà més de les que s'han demanat).

La conclusió és que **sempre**, quan es llegeix d'una connexió TCP, s'ha de comprovar la quantitat de bytes rebuts, i comparar-ho amb la quantitat desitjada per saber si cal fer una nova lectura.

Quan d'un socket es llegeixen zero bytes és senyal que la connexió s'ha tancat.

7.7.1 Funcions *send* i *recv*

Estan declarades de la següent manera:

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

ssize_t send (int fd, const void *buff, size_t len, int flags);
ssize_t recv (int fd, void *buff, size_t len, int flags);
```

Els tres primers paràmetres són idèntics als tres primers paràmetres de `read` i `write`, respectivament. La novetat està en el quart paràmetre. Consisteix en una màscara de bits, on cadascun d'ells és un flag amb un determinat significat. Els flags es poden combinar amb l'operador d'*or* binària del llenguatge C. Hi ha bastants flags, aquí només es comentaran els que es consideren més rellevants:

`MSG_DONTWAIT` Fa que aquesta — i només aquesta — operació sigui no bloquejant, és a dir, com si el socket fos no bloquejant. Això vol dir que:

1. Per a les escriptures, s'escriuran tants bytes com càpiguen al buffer del sistema operatiu. Si en caben menys, se n'escriuran menys. I si no en caben, llavors retornarà -1 i l'error serà del tipus "try again".
2. Per a les lectures, llegirà els bytes que hi hagi al buffer del sistema operatiu. Si no n'hi ha, llavors retornarà -1 i l'error serà del tipus "try again".

La idea d'un socket no bloquejant és que, com el seu nom indica, el procés no es quedi bloquejat esperant que es completi l'operació.

`MSG_MORE` Per a `send`, indica que s'ha d'obtenir el mateix efecte que l'opció de socket `TCP_CORK`. Aquest flag no és vàlid per a `recv`.

`MSG_PEEK` Per a `recv`, indica que els bytes llegits s'han de quedar al buffer del sistema operatiu, de manera que la següent lectura llegirà les mateixes dades. Aquest flag no és vàlid per a `send`.

`MSG_WAITALL` Per a `recv`, indica que el procés s'ha de quedar bloquejat fins que es rebin tots els bytes demanats. Ara bé, fins i tot amb aquest flag es poden rebre menys bytes dels demanats, en alguna de les següents circumstàncies:

1. S'ha rebut un senyal.
2. Es produeix un error (per exemple, es rep un segment de reset).
3. La connexió s'han tancat i es llegeixen els últims bytes que hi havia pendents de llegir.

Aquest flag no és vàlid per a `send`.

Un últim apunt, sobre el flag `MSG_PEEK`. Hi ha una manera de saber els bytes que queden per llegir en un socket, amb un tipus de `ioctl` anomenat `FIONREAD`. Per a més detalls, es pot consultar la pàgina manual de `ioctl`.

7.8 Exemple

El següent exemple consisteix en un servidor que accepta connexions i, un cop oberta la connexió, llegeix dades i les mostra per pantalla. El client, al seu torn, es connecta al servidor, envia una salutació i finalitza la seva execució.

El codi del servidor és el següent:

```
1  #include <arpa/inet.h>
2  #include <netinet/in.h>
3  #include <sys/socket.h>
4  #include <stdint.h>
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <strings.h>
9  #include <unistd.h>
10
11 int
12 main (int argc, char *argv[])
13 {
14   // comprovem que se'ns hagi passat el port
15   if (argc < 2)
16     {
17       fprintf (stderr, "Error: falta especificar el port\n");
18       exit (EXIT_FAILURE);
19     }
20
21   // comprovem la validesa del port
22   uint16_t port;
23   int aux = atoi (argv[1]);
24   if (aux < 1 || aux > 65535)
25     {
26       fprintf (stderr, "Error: %s  s un port inv lid\n", argv[1]);
27       exit (EXIT_FAILURE);
28     }
29   port = aux;
```



```
30
31    // creem el socket
32    int sockfd;
33    sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
34    if (sockfd < 0)
35      {
36        perror ("socket TCP");
37        exit (EXIT_FAILURE);
38      }
39
40    // especifiquem l'adre a que volem vincular al nostre socket
41    // admetrem connexions dirigides a qualsevol IP de la nostra m quina
42    // al port especificat per l nia de comandes
43    struct sockaddr_in s_addr;
44    bzero (&s_addr, sizeof (s_addr));
45    s_addr.sin_family = AF_INET;
46    s_addr.sin_port = htons (port);
47    s_addr.sin_addr.s_addr = INADDR_ANY;
48
49    // al cridar bind, hem de fer un cast:
50    // bind espera struct sockaddr *
51    // i nosaltres passem struct sockaddr_in *
52    if (bind (sockfd, (void *) &s_addr, sizeof (s_addr)) < 0)
53      {
54        perror ("bind");
55        exit (EXIT_FAILURE);
56      }
57
58    // ara obrim el port, 5  s un valor t pic
59    listen (sockfd, 5);
60
61    while (1)
62      {
63        struct sockaddr_in c_addr;
64        socklen_t c_len = sizeof (c_addr);
65
66        // al cridar accept hem de fer el mateix cast que per a bind,
67        // i pel mateix motiu
68        int newsock = accept (sockfd, (void *) &c_addr, &c_len);
69        if (newsock < 0)
70          {
71            perror ("accept");
72            exit (EXIT_FAILURE);
73          }
74        printf ("Nova connexi  de %s:%hu\n",
75                inet_ntoa (c_addr.sin_addr), ntohs (c_addr.sin_port));
76
77        // mentre vinguin dades del client,
78        // les llegim i les mostrem per pantalla
```

```
79        ssize_t len;
80        char buff[513];           // un espai per \0, si calgu s
81        do
82          {
83            len = read (newsock, buff, 512);
84            buff[len] = 0;
85            printf ("%s\n", buff);
86          }
87        while (len > 0);
88        close (newsock);
89      }
90
91    return EXIT_SUCCESS;
92  }
```

El codi del client és el següent:

```
1   #include <arpa/inet.h>
2   #include <netinet/in.h>
3   #include <sys/socket.h>
4   #include <errno.h>
5   #include <stdint.h>
6   #include <stdio.h>
7   #include <stdlib.h>
8   #include <string.h>
9   #include <strings.h>
10  #include <unistd.h>
11
12  int
13  main (int argc, char *argv[])
14  {
15    // comprovem que se'ns hagi passat l'adre a IP
16    // i el port TCP
17    if (argc < 3)
18      {
19        fprintf (stderr, "Error: falten arguments\n");
20        exit (EXIT_FAILURE);
21      }
22
23    // comprovem la validesa del port
24    uint16_t port;
25    int aux = atoi (argv[2]);
26    if (aux < 1 || aux > 65535)
27      {
28        fprintf (stderr, "Error: %s  s un port inv lid\n", argv[1]);
29        exit (EXIT_FAILURE);
30      }
31    port = aux;
32
33    // comprovem la validesa de l'adre a IP
```



```
34    // i la convertim a format binari
35    struct in_addr ip_addr;
36    if (inet_aton (argv[1], &ip_addr) == 0)
37      {
38        fprintf (stderr, "inet_aton (%s): %s\n", argv[1], strerror (errno));
39        exit (EXIT_FAILURE);
40      }
41
42    // creem el socket
43    int sockfd;
44    sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
45    if (sockfd < 0)
46      {
47        perror ("socket TCP");
48        exit (EXIT_FAILURE);
49      }
50
51    // especifiquem l'adre a del servidor
52    struct sockaddr_in s_addr;
53    bzero (&s_addr, sizeof (&s_addr));
54    s_addr.sin_family = AF_INET;
55    s_addr.sin_port = htons (port);
56    s_addr.sin_addr = ip_addr;
57
58    // i ara ja ens podem connectar
59    // al cridar connect, hem de fer un cast:
60    // bind espera struct sockaddr *
61    // i nosaltres passem struct sockaddr_in *
62    if (connect (sockfd, (void *) &s_addr, sizeof (s_addr)) < 0)
63      {
64        perror ("connect");
65        exit (EXIT_FAILURE);
66      }
67
68    char msg[] = "Hola!";
69    // cridem sizeof i no strlen perqu  msg  s un array
70    // cridar strlen tamb  seria correcte
71    write (sockfd, msg, sizeof (msg));
72
73    return EXIT_SUCCESS;
74  }
```

Capítol 8

Threads

8.1 Conceptes

Un thread és una subdivisió d'un procés, que s'executa al mateix temps que la resta del procés. En el fons, threads i processos vénen a representar el mateix concepte (i Linux els tracta, internament, de la mateixa manera, tenint en compte les seves diferències). La diferència rau en què un thread és una subdivisió d'un procés, en què tots els threads d'un procés ho comparteixen gairebé tot i en què no hi ha relació jeràrquica entre ells (no podem parlar de "threads pare" i "threads fill").

Si els hem deixat pel final és perquè són una eina molt difícil de programar correctament. Són tremendament útils i molt utilitzats (solucionen eficientment alguns problemes dels processos), però pel fet de compartir tantes coses, donen moltes oportunitats a cometre errors. Errors que, en la majoria dels casos, són difícils de solucionar i encara molt més difícils de trobar, perquè acostumen a exhibir-se de forma aleatòria.

Aquest capítol ens servirà d'introducció als aspectes més importants relacionats amb els threads. Per a alumnes interessats en aprofundir sobre el tema, es recomanen els capítols 29, 30, 31, 32 i 33 (pàgines 617–698) de Kerrisk [2010] i el capítol 26 (pàgines 675–708) de Stevens et al. [2004].

8.2 Creació de threads

8.2.1 Exemple

Comencem vegent el següent exemple:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static void
errExitEN (int s, const char *msg)
{
  printf ("%s: %s\n", msg, strerror (s));
  exit (EXIT_FAILURE);
}

static void *
threadFunc (void *arg)
{
  char *s = (char *) arg;

  printf ("%s", s);

  return (void *) strlen (s);
}

int
main (void)
{
  pthread_t t1;
  void *res;
  int s;

  s = pthread_create (&t1, NULL, threadFunc, "Hello world\n");
  if (s != 0)
    errExitEN (s, "pthread_create");

  printf ("Message from main()\n");
  s = pthread_join (t1, &res);
  if (s != 0)
    errExitEN (s, "pthread_join");

  printf ("Thread returned %ld\n", (long) res);

  exit (EXIT_SUCCESS);
}
```

Resumint, aquest programa crea un thread, que executa la funció *threadFunc*. Quan aquesta funció finalitza, el thread s'acaba i desapareix. La funció *errExitEN* es fa servir tan sols per si cal mostrar missatges d'error. La sortida del programa és la següent:


```
1 Message from main()
2 Hello world
3 Thread returned 12
```

Important Tots els programes amb threads cal compilar-los passant l'opció *-lpthread* al final de tot del *gcc*:

```
gcc -std=gnu99 simple_thread.c -lpthread
```

8.2.2 La funció *pthread_create*

Si mirem el codi font de l'exemple, veurem una funció anomenada *pthread_create*. Aquesta funció s'encarrega de crear un thread, i rep quatre arguments:

1. El primer és un punter a una variable de tipus *pthread_t*, que ve a ser un identificador de thread.
2. A l'exemple anterior, el segon argument és `NULL`. Això vol dir que el thread es crearà amb els atributs per defecte. De moment no cal entrar en més detalls.
3. El tercer argument és el nom de la funció que executa el thread.
4. El quart argument és l'argument que rep la funció que executa el thread. Pot ser un punter `NULL`.

Cal tenir en compte un detall molt important: la funció que executa el thread ha de tenir el següent prototipus:

```
void * funcio (void * arg);
```

És a dir, ha de retornar "`void *`" i ha de rebre un paràmetre de tipus "`void *`".

Prototipus El prototipus de la funció *pthread_create* és el següent

```
#include <pthread.h>

int pthread_create (pthread_t *thread,
                    const pthread_attr_t *attr,
                    void *(*start_routine) (void *),
                    void *arg);
```

8.2.3 Diferències amb *fork*

Val la pena comentar algunes diferències entre la funció que crea processos (*fork*) i la que crea threads (*pthread_create*):

1. La funció *fork* no rep cap paràmetre. La funció *pthread_create* en rep quatre.

2. Quan es crea un nou procés, tant el pare com el fill segueixen l'execució a continuació de la crida a *fork*. En canvi, un thread executa una funció en particular, i quan aquesta funció retorna el thread finalitza i desapareix.

3. Entre els threads d'un procés no hi ha relació de pares i fills, tots els threads són "germans" entre ells.

4. La funció *fork* retorna diferent segons sigui el pare o el fill: retorna zero pel fill; al pare retorna l'identificador del procés. En canvi, la funció *pthread_create* retorna zero si el thread s'ha aconseguit crear, i un enter difernt de zero en cas d'error. El thread que s'ha creat comença a executar la funció que s'ha passat com a argument a la funció *pthread_create*.

8.2.4 La funció *pthread_join*

A l'exemple amb què hem començat, hi ha una crida a una funció anomenada *pthread_join*. Aquesta funció és l'equivalent (més o menys) de la funció *wait* (i similars) aplicada als threads. Fa dues feines:

1. El thread que la crida es bloqueja, esperant que finalitzi el thread indicat. En el cas de l'exemple, el thread principal s'espera que s'acabi el thread creat anteriorment.

2. Es recull el valor de retorn del thread. Ja hem dit que les funcions que executen threads han de retornar un punter.

Si ens fixem en l'exemple, veurem que s'ha declarat una variable *res*, que és un punter, i al cridar *pthread_join* es passa un punter a aquest punter. Això, a part de confondre una mica el programador que no hi està acostumat, té una conseqüència molt important: les funcions que executen threads han de retornar punters a zones de memòria que no desapareguin quan el thread finalitzi. Per exemple, no poden retornar punters a variables locals (excepte si la variable local és estàtica, és a dir, declarada amb *static*).

En el cas de l'exemple s'ha fet una petita "marranada", i és que en realitat la funció que executa el thread (*threadFunc*) no retorna un punter, sinó un enter: la longitud de la cadena rebuda com a argument. Fent els *casts* necessaris, això és perfectament vàlid.

Prototipus El prototipus de la funció *pthread_join* és el següent:

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

Important No tots els threads admeten la funció *pthread_join*, és a dir, hi ha threads pels quals un no es pot esperar que s'acabin. En aquest capítol no veurem com es fa. També és important remarcar que, dins d'un mateix procés, qualsevol thread pot esperar la finalització de qualsevol thread, ja que no hi ha jerarquia entre els threads (tipus "pares i fills"). A més a més, al cridar la funció *pthread_join* cal especificar, obligatòriament, el thread pel qual ens volem esperar — no es pot esperar la finalització de *qualsevol* thread.

8.3 Sincronització de threads

8.3.1 Introducció

A l'introduir els threads hem dit que tots els threads d'un procés comparteixen el seu espai de memòria. És a dir, per definició, tota la memòria és compartida. La qual cosa vol dir que hem d'estar molt atents a possibles conflictes i col·lisions. La qual cosa vol dir que una aplicació amb threads tindrà una bona quantitat de semàfors (i d'altres mecanismes similars).

En aquest apartat no explicarem la teoria que hi ha al darrere de per què cal fer servir semàfors, ni què és una regió crítica; assumim que el lector coneix aquests conceptes. Explicar-los està fora de l'abast d'aquest llibre i ens ocuparia massa espai, que volem dedicar a les eines que els threads ens ofereixen.

8.3.2 Exemple 1

Per a explicar aquestes eines, veurem un primer exemple d'una aplicació que conté dos threads. Cadascun d'aquests threads executa una funció que incrementa una variable global i, al final, mostra el seu valor.

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  static int glob = 0;
7
8  static void
9  errExitEN (int s, const char *msg)
10 {
11   printf ("%s: %s\n", msg, strerror (s));
12   exit (EXIT_FAILURE);
```

```
13  }
14
15  static void *
16  threadFunc (void *arg)
17  {
18    int loops = *((int *) arg);
19    int loc, j;
20
21    for (j = 0; j < loops; j++)
22      {
23        loc = glob;
24        loc++;
25        glob = loc;
26      }
27
28    return NULL;
29  }
30
31  int
32  main (int argc, char *argv[])
33  {
34    pthread_t t1, t2;
35    int loops, s;
36
37    loops = (argc > 1) ? atoi (argv[1]) : 10000000;
38
39    s = pthread_create (&t1, NULL, threadFunc, &loops);
40    if (s != 0)
41      errExitEN (s, "pthread_create");
42    s = pthread_create (&t2, NULL, threadFunc, &loops);
43    if (s != 0)
44      errExitEN (s, "pthread_create");
45
46    s = pthread_join (t1, NULL);
47    if (s != 0)
48      errExitEN (s, "pthread_join");
49    s = pthread_join (t2, NULL);
50    if (s != 0)
51      errExitEN (s, "pthread_join");
52
53    printf ("glob = %d\n", glob);
54    exit (EXIT_SUCCESS);
55  }
```

Com és d'esperar, si el lector compila aquest exemple i l'executa diferents vegades, veurà que amb cada execució el resultat és diferent. L'eina que els threads ens ofereixen per a solucionar aquest problema es diuen *mutex* (de "mutual exclusion"); es tracta de semàfors binaris.

8.3.3 Exemple 2

A continuació veurem el mateix exemple però solucionat amb mutexs:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

static int glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static void
errExitEN (int s, const char *msg)
{
  printf ("%s: %s\n", msg, strerror (s));
  exit (EXIT_FAILURE);
}

static void *
threadFunc (void *arg)
{
  int loops = *((int *) arg);
  int loc, j, s;

  for (j = 0; j < loops; j++)
    {
      s = pthread_mutex_lock (&mtx);
      if (s != 0)
        errExitEN (s, "pthread_mutex_lock");

      loc = glob;
      loc++;
      glob = loc;

      s = pthread_mutex_unlock (&mtx);
      if (s != 0)
        errExitEN (s, "pthread_mutex_unlock");
    }

  return NULL;
}

int
main (int argc, char *argv[])
{
  pthread_t t1, t2;
  int loops, s;

  loops = (argc > 1) ? atoi (argv[1]) : 10000000;
```

```
47
48    s = pthread_create (&t1, NULL, threadFunc, &loops);
49    if (s != 0)
50      errExitEN (s, "pthread_create");
51    s = pthread_create (&t2, NULL, threadFunc, &loops);
52    if (s != 0)
53      errExitEN (s, "pthread_create");
54
55    s = pthread_join (t1, NULL);
56    if (s != 0)
57      errExitEN (s, "pthread_join");
58    s = pthread_join (t2, NULL);
59    if (s != 0)
60      errExitEN (s, "pthread_join");
61
62    printf ("glob = %d\n", glob);
63    exit (EXIT_SUCCESS);
64  }
```

8.3.4 Explicació

Els elements fonamentals de la solució són:

- La variable *mtx*, de tipus `pthread_mutex_t`. Aquesta variable és el mutex.
- La funció *pthread_mutex_lock*, que es crida a l'inici de la regió crítica.
- La funció *pthread_mutex_unlock*, que es crida al final de la regió crítica.

Un mutex és un element que s'utilitza per a protegir regions crítiques. Quan un thread vol entrar en una regió crítica protegida per un mutex, crida la funció *pthread_mutex_lock*, que serveix per "demanar permís" per entrar a la regió. Si no hi ha cap altre thread a la regió, aquesta funció permet que el thread entri a la regió crítica, i al mateix temps impedirà que cap altre thread hi entri. Si ja hi havia un thread executant la regió crítica, llavors la funció *pthread_mutex_lock* adormirà el thread.

Més tard, quan el thread vol sortir de la regió crítica crida *pthread_mutex_unlock*, per tal que, si algun altre thread vol entrar a la regió crítica, hi pugui entrar; i si hi havia threads adormits esperant poder-hi entrar, es despertin i un d'ells (només un) hi pugui entrar.

A la resta d'apartats anirem explicant, en detall, quins passos cal seguir per tal de crear i destruir els mutexs i quin és el seu ús correcte.

8.3.5 Creació de mutexs

A l'exemple solucionat hi ha una nova variable global, *mtx*, declarada de la següent manera:

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

No n'hi ha prou amb crear la variable *mtx* de tipus *pthread_mutex_t*: cal inicialitzar-la. L'exemple ha optat per fer-ho d'aquesta manera; es podria haver fet servir la funció *pthread_mutex_init*:

```
pthread_mutex_init (&mtx, NULL);
```

on el primer argument és el punter a la variable de tipus *pthread_mutex_t* i el segon argument és un punter als atributs del mutex (és opcional, i és vàlid passar un punter `NULL`. En aquest capítol no parlarem dels atributs dels mutexs.

Quan es pot fer servir *PTHREAD_MUTEX_INITIALIZER*? Només quan es tracta de variables globals, o variables locals declarades *static*.

8.3.6 Destrucció de mutexs

Els mutexs que s'han creat amb *pthread_mutex_init* s'han de destruir, quan no hi hagi cap thread que els estigui fent servir. Per a tal fi disposem de la funció *pthread_mutex_destroy*, que rep com a únic argument un punter al mútex que volem destruir.

8.3.7 Bloqueix (o adquisició) d'un mutex

El fet de cridar la funció *pthread_mutex_lock* sobre un mutex s'anomena *bloquejar* el mutex, ja que es bloqueja l'accés a la regió crítica per a la resta de threads. A la solució que hem vist anteriorment, ens podem adonar que el thread, quan vol entrar a la regió crítica, bloqueja el mutex. Aquest mutex té, per finalitat, protegir la regió crítica, per això els threads bloquegen el mutex abans d'entrar-hi. Només un thread pot estar a la regió crítica. Per això, *pthread_mutex_lock* adorm els threads quan ja n'hi ha un a la regió crítica. És un error (i pot portar conseqüències greus) intentar bloquejar dues (o més) vegades seguides un mateix mutex.

Hi ha dues variants d'aquesta funció:

- La funció *pthread_mutex_trylock* no adorm mai el thread. Si el mutex ja està bloquejat, no retorna zero sinó un codi d'error indicant que el mutex ja està bloquejat.
- La funció *pthread_mutex_timedlock* adorm el thread si el mutex està bloquejat, però només durant una quantitat màxima de temps.

Aquestes dues variants s'utilitzen molt menys que no pas *pthread_mutex_lock*.

8.3.8 Desbloqueig d'un mutex

El fet de cridar la funció *pthread_mutex_unlock* sobre um mutex s'anomena *desbloquejar* el mutex. Quan un thread surt de la regió crítica, ha de cridar aquesta funció per deixar pas als altres threads que hi volen entrar; a més a més, al tornar-hi a entrar i tornar a cridar *pthread_mutex_lock* el thread podria adormir-se eternament.

Hi ha dues normes importants a tenir en compte pel que fa el desbloqueig de mutexs:

1. Un thread només pot desbloquejar un mutex que ell mateix hagi bloquejat abans.

2. Un thread només pot desbloquejar un mutex que estigui bloquejat — dit d'una altra manera, no es pot desbloquejar dues vegades seguides (o més) un mutex.

PART II

Conceptes avançats

Capítol 9

Processos en UNIX: conceptes avançats

9.1 Introducció

El primer capítol d'aquest llibre va presentar el concepte de procés i va explicar els mecanismes bàsics per a la seva creació i finalització, a més d'altres eines com ara la funció `wait`. En aquest capítol es pretén completar el material presentat llavors i ampliar-lo per tal d'aprofundir en el concepte de procés i en les eines que el sistema UNIX ofereix relacionades amb el control de processos.

9.2 La funció *vfork*

Quan al capítol 1 es va introduir el concepte de procés, es va dir explícitament que, al crear un nou procés, el pare i el fill no comparteixen cap zona de memòria, excepte (en tot cas) la de codi i aquelles zones creades explícitament per ser compartides. Les primeres implementacions de la funció *fork* copiaven absolutament totes les zones de memòria que no havien de ser compartides. Aquest enfocament és correcte, tot i que molt ineficient, especialment tenint en compte que la majoria de fills es fan per a executar un nou programa. Quan s'executa un nou programa, tota la memòria del procés es descarta per tal de carregar un nou programa. Per tant, doncs, la còpia feta al cridar `fork` esdevenia del tot inútil.

A tal fi es va crear la funció `vfork`:

```
#include <unistd.h>
```

```
pid_t vfork (void);
```

Aquesta funció actua com `fork`, però amb les següents diferències:

- No es copia absolutament cap zona de memòria, de manera que pare i fill comparteixen la memòria fins que el fill mor o bé executa alguna de les funcions de la família `exec` (vegi's l'apartat 1.5).
- L'execució del pare se suspèn fins que el fill mor o executa alguna de les funcions de la família `exec`.

La finalitat d'aquesta funció és millorar enormement l'eficiència a l'hora d'executar nous programes. La diferent semàntica entre `fork` i `vfork` fa que calgui anar amb compte amb què fa el fill entre que és creat i executa el nou programa. Tota la memòria és compartida, i per tant quasi tot el que faci afectarà el pare. Pot operar sobre file descriptors, però és molt perillós que cridi la funció `exit`, ja que aquesta funció tanca tots els `FILE *` oberts. Per això convé que cridi la funció `_exit` per a finalitzar la seva execució, ja que aquesta funció no tanca cap `FILE *`.

Actualment, la funció `vfork` és pràcticament innecessària, ja que les implementacions actuals de `fork` no copien zones de memòria fins que és necessari. És a dir, pare i fill comparteixen tota la memòria fins que algun d'ells hi escriu; en aquest cas es fa la còpia. D'aquesta política se'n diu *copy-on-write*.

9.3 La funció *_exit*

Per a finalitzar explícitament un procés, en la gran majoria de casos és suficient cridar la funció `exit`. Hi ha alguns casos, però, en què les accions que duu a terme aquesta funció poden produir efectes *inesperats*, com els de l'exemple següent:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int
6  main (void)
7  {
8    printf ("Hello, world!\n");
9    write (1, "Bye!\n", 5);
10
11   if (fork () == -1)
12     perror ("fork");
13
14   exit (0);
15 }
```

Si s'executa aquest programa, s'obté la següent sortida:


```
$ gcc fork_stdio.c
$ ./a.out
Hello, world!
Bye!
$
```

En canvi, si s'executa d'aquesta manera, s'obté una altra sortida:

```
$ ./a.out > out.txt
$ cat out.txt
Bye!
Hello, world!
Hello, world!
$
```

Aquest efecte té a veure, per una banda, amb el *buffering* dels `FILE *` (vegi's l'apartat 10.4.2) i el fet que `exit` buida els buffers dels `FILE *` abans de tancar-los. Es proposa a l'alumne explicar detalladament aquests efectes i proposar diferents solucions per tal que:

1. El missatge "Hello, world!" aparegui abans que "Bye!".
2. El missatge "Hello, world!" no aparegui duplicat.

9.4 La funció *clone*

9.4.1 Introducció

La funció `clone` és específica de Linux i, com `fork` i `vfork`, serveix per a crear nous processos. Ara bé, la funció `clone` permet un control molt més ampli i precís que qualsevol altre mètode per a crear processos; la seva principal finalitat és implementar les llibreries de *threads*. Per norma general, els programadors no haurien de fer servir la funció `clone`: en primer lloc perquè no és *portable* (només existeix en Linux); en segon lloc perquè en la immensa majoria de casos les seves necessitats (del programador) se satisfan perfectament amb `fork` o *threads*. Cal tenir en compte que la funció `clone` és molt propera al nucli del sistema operatiu — en el sentit que permet controlar aspectes referents a com el nucli del sistema operatiu implementa els processos i en permet el seu funcionament.

La funció `clone` està declarada de la següent manera:

```
#define _GNU_SOURCE
#include <sched.h>

int clone (int (*func)(void *), void *child_stack, int flags,
           void *func_arg, ... /* pid_t *pid,
           struct user_desc *tls, pid_t ctid */);
```

Un procés nou creat amb `clone` és gairebé una còpia exacta del seu pare, igual que passa amb `fork`. Ara bé, el fill no comença la seva execució a partir del codi que hi ha després de la crida a `clone`, sinó que es crida la funció apuntada pel paràmetre `func` (a aquesta funció li direm *funció fill*). A aquesta funció se li passa el paràmetre `func_arg`.[1]

L'execució del fill s'acaba quan la funció `func` retorna o quan crida `exit` o `_exit`. El procés pare pot esperar que el fill finalitzi la seva execució amb `wait` o equivalent.

L'argument `child_stack` apunta a una zona de memòria ubicada anteriorment, que servirà de segment de pila per al procés fill. Cal tenir en compte que en la majoria de les arquitectures de hardware, la pila creix "cap avall" — és a dir, l'inici de la pila correspon a una adreça de memòria més alta que el seu final. L'argument `flags` té una doble finalitat:

1. El byte de menys pes indica el senyal que ha de rebre el pare quan el procés fill finalitzi la seva execució. Si val zero, el pare no rep cap senyal.

2. La resta de bytes són una màscara de bits que controla diferents detalls de la creació del fill. N'hi ha bastants, els més rellevants dels quals se'n parla a l'apartat 9.4.2.

De la resta dels arguments se'n parlarà quan surtin en la discussió sobre els diferents flags, a l'apartat següent. El valor de retorn de `clone` és el TID (*thread identifier*) del fill, o bé -1 si hi ha hagut algun error.

9.4.2 Flags

Advertència Abans de començar a detallar alguns dels diferents flags que admet la funció `clone`, val la pena copiar l'advertiment que es pot trobar a la pàgina 603 de Kerrisk [2010]:

> At this point, it is worth remarking that, to some extent, we are playing with words when trying to draw a distinction between the terms *thread* and *process*. It helps a little to introduce the term *kernel scheduling entity* (KSE), which is used in some texts to refer to the objects that are dealt with by the kernel scheduler. Really, processes and threads are simply KSEs that provide for greater and lesser degrees of sharing of attributes (virtual memory, open file descriptors, signal dispositions, process ID, and so on) with other KSEs. The POSIX threads specification provides just one out of various possible definitions of which attributes should be shared between threads.

[1] En realitat, la crida al sistema `sys_clone`, implementada pel nucli del sistema operatiu, no té els paràmetres `func` i `func_arg`; el fill comença la seva execució a partir de la crida a `clone`, igual que passa amb la funció `fork`. Ara bé, la implementació de la llibreria de C de la funció `clone` fa que, just quan retorna `clone`, el fill cridi la funció `func`, passant-li el paràmetre `func_arg`.

CLONE_FILES Si aquest flag és present, pare i fill comparteixen la mateixa taula de descriptors oberts (per defecte no la comparteixen). Què significa que comparteixin la taula de descriptors oberts? Doncs que si un procés crida (per posar exemples) `open`, `close`, `dup`, `socket`, etc, l'altre procés també ho veurà. Quan es crea un procés amb `fork`, a partir de la creació del fill tot el que faci un procés és privat, és a dir, si un dels dos tanca un arxiu, a l'altre procés l'arxiu segueix obert. Cal remarcar que es tracta de què passa un cop el fill ja s'ha creat, no de què passa en el moment de crear el fill.

Els processos creats amb `fork` no comparteixen la taula de descriptors oberts; els threads sí la comparteixen.

CLONE_FS Si aquest flag és present, pare i fill comparteixen la màscara de permisos (*umask*), el directori arrel i el *current working directory*. Per tant, si un procés (pare o fill) canvia algun d'aquests atributs, a l'altre procés també queden canviats. Els processos creats amb `fork` no comparteixen aquests atributs; els threads sí els comparteixen.

CLONE_SIGHAND Si aquest flag és present, pare i fill comparteixen la disposició dels senyals — és a dir, els dos comparteixen com reaccionen als senyals. Per exemple, si un procés (pare o fill) decideix ignorar un senyal, o instal·lar-hi un *signal handler*, aquest canvi també passa a ser efectiu a l'altre procés. Ara bé, la màscara (conjunt) de senyals bloquejats i la de senyals pendents sempre és independent per a cada procés.

Els processos creats amb `fork` no comparteixen la disposició dels senyals; els threads sí la comparteixen.

A partir de la versió 2.6 de Linux, si aquest flag és present, també ho ha de ser el flag `CLONE_VM`.

CLONE_VM Si aquest flag és present, pare i fill comparteixen l'espai de memòria virtual — és a dir, comparteixen variables globals, memòria demanada amb `malloc` i `mmap`, etc. Aquesta és la propietat definitòria dels threads. Els processos creats amb `fork` no comparteixen la memòria virtual.

CLONE_VFORK Si aquest flag és present, l'execució del pare queda aturada fins que el fill finalitza la seva execució. Els threads i els processos creats amb `fork` no tenen aquest flag present; en canvi, els processos creats amb `vfork` sí la tenen — d'aquí el nom del flag.

CLONE_PTRACE Si el procés pare no està sent debugat, llavors aquest flag és innocu. Ara bé, si el pare està sent debugat i aquest flag és present, llavors el fill també

serà debugat. Per a entendre més exactament aquest concepte, convé conèixer com funcionen els debuggers i com s'ho fan per debugar processos que creen fills.[2]

CLONE_PARENT Suposem un procés A que crea un nou procés B. El procés A és, al seu torn, fill d'un procés P. Per defecte, el procés B tindrà com a pare el procés A. Si el flag `CLONE_PARENT` és present, llavors B tindrà com a pare el procés P.

CLONE_THREAD Si el flag és present, llavors al fill se'l col·loca al mateix grup de threads que el pare. En cas contrari, se l'ubica a un nou grup de threads. El concepte de grup de threads es va crear per tal de complir el requeriment que tots els threads d'un procés han de tenir el mateix PID — és a dir, que la funció `getpid` ha de retornar el mateix valor per a tots els threads (d'un mateix procés). És a dir, un grup de threads és un grup de KSEs (*Kernel Scheduling Entity*) que tenen el mateix identificador de grup de threads (TGID). En realitat, doncs, la funció `getpid` retorna el TGID, que per a un procés d'un sol thread és el mateix que el PID. Cada thread s'identifica amb el TID. Un thread pot obtenir el seu TID amb la crida al sistema `gettid`.[3] Cal tenir en compte que aquests TIDs no tenen res a veure amb els *thread id* de la llibreria POSIX threads (`pthread_t`).

Tots els threads d'un grup de threads tenen el mateix procés pare. Quan tots els threads han mort, llavors el pare rebrà el senyal especificat per a notificar la mort d'un fill. Ara bé, quan un thread s'ha creat amb el flag `CLONE_THREAD` present, llavors el pare no rep cap senyal, i no pot emprar la funció `wait` (o equivalent). Per a detectar la finalització d'un thread cal emprar una eina especial de sincronització anomenada *futex*.[4]

Si un thread d'un grup de threads executa `exec` (alguna de les funcions de la família), llavors tots els threads del grup finalitzen la seva execució, i el nou procés s'executa en el thread del "líder" del grup (aquell thread en què el seu TID és igual al TGID). A més a més, el senyal a enviar al pare en cas de mort torna a ser `SIGCHLD`.

Si un thread d'un grup de threads crea un procés amb `fork` o `vfork`, llavors qualsevol altre thread del grup el pot monitoritzar amb `wait` (o equivalent).

CLONE_PARENT_SETTID, CLONE_CHILD_SETTID, CLONE_CHILD_CLEARTID Aquests tres flags serveixen per a possibilitar la implementació dels threads amb els requisits que marca l'especificació POSIX, i estan relacionats amb els arguments `ptid` i `ctid` de la funció `clone`.

[2] Pel que fa al debugger *gdb*, l'usuari pot seleccionar si, quan un procés que està sent debugat crea un fill, si es vol seguir debugant el pare o es vol debugar el fill; en Linux es pot debugar els dos conjuntament.

[3] No hi ha cap funció per a cridar directament aquesta crida al sistema; cal usar la funció *syscall*. La pàgina manual d'aquesta funció ofereix, precisament, un exemple sobre com cridar *gettid*

[4] Aquesta eina de sincronització no està pensada per al programador habitual, sinó que s'empra per a la implementació de mutexs i d'altres que sí estan pensades per a la majoria de programadors. Es pot consultar la pàgina manual de *futex* i un *paper* d'Ulrich Drepper curiosament titulat "Futexes are tricky".

Si el flag `CLONE_PARENT_SETTID` és present, llavors `clone` copia el TID (*thread ID*) a la zona de memòria apuntada per `ptid`, **ababns** que es dupliqui la memòria (cas de duplicar-se). D'aquesta manera, tant el pare com el fill poden veure el TID del fill independentment de si el flag `CLONE_VM` és present. Es pot objectar que la pròpia funció `clone` retorna el TID, però el mecanisme ofert per aquest flag evita *condicions de carrera* que es podrien produir (vegi's l'explicació de la pàgina 606 de Kerrisk [2010] per a un exemple).

Si el flag `CLONE_CHILD_SETTID` és present, llavors `clone` copia el TID a la zona de memòria apuntada per `ctid`. Aquesta còpia es fa només al fill, de manera que no afecta el pare — excepte si el flag `CLONE_VM` és present.

Si el flag `CLONE_CHILD_CLEARTID` és present, llavors la zona de memòria apuntada per `ctid` es posa a zero quan el fill finalitza la seva execució.

L'ús d'aquests flags, combinat amb un mecanisme anomenat *futex*, permet implementar la notificació de finalització d'un *thread*, requerida per la funció `pthread_join` ("similar" a la funció `wait`). Quan es crea un thread amb `pthread_create`, es crida `clone` fent que `ptid` i `ctid` apuntin a la mateixa zona de memòria, amb els flags `CLONE_PARENT_SETTID` i `CLONE_CHILD_CLEARTID` presents. Llavors, quan el thread finalitza i la zona de memòria apuntada per `ctid` es posa a zeros, aquest canvi es fa visible a tots els threads del procés (ja que al crear un thread també s'activa el flag `CLONE_VM`). El nucli del sistema operatiu tracta aquella zona de memòria com un *futex*, que és un mecanisme de sincronització. La funció `pthread_join` utilitza el futex per esperar que la zona de memòria es posi a zero. Quan això passa, el sistema operatiu desperta els threads que estiguessin esperant aquest canvi — justament el thread que ha cridat `pthread_join`.

`CLONE_SETTLS` Aquest flag té a veure amb un concepte anomenat *thread-local storage*, del qual se'n parlarà a l'apartat ***. Si el flag és present, llavors l'argument `tls` de la funció `clone` apunta a un descriptor de *thread-local storage*.

Exemples d'ús La funció `fork` crea un procés d'una forma similar a com ho faria la funció `clone` amb els següents flags:

```
SIGCHLD
```

mentre que a la funció `vfork` li correspondrien els següents flags:

```
CLONE_VM | CLONE_VFORK | SIGCHLD
```

En Linux hi ha dues implementacions de threads: LinuxThreads i NPTL (*Native POSIX Threading Library*). La primera crea els threads amb els següents flags:

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND
```

i la segona (NPTL):

```
CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND | CLONE_THREAD |
CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID |
CLONE_SYSVSEM
```

9.4.3 Exemple

Com a exemple, es presenta una variació de l'exemple ofert per Michael Kerrisk a la web del seu llibre Kerrisk [2010]. S'ha modificat el codi font per tal que consti d'un sol arxiu de codi font.[5]

```
1  #define _GNU_SOURCE
2  #include <sys/stat.h>
3  #include <sys/wait.h>
4  #include <errno.h>
5  #include <fcntl.h>
6  #include <sched.h>
7  #include <signal.h>
8  #include <stdbool.h>
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #include <unistd.h>
13
14 #ifndef CHILD_SIG
15 #define CHILD_SIG SIGUSR1       /* Signal to be generated on termination
16                                    of cloned child */
17 #endif
18 #define STACK_SIZE (64 * 1024)
19 #define START_UMASK S_IWOTH
20
21 /* For passing info to child startup function */
22 typedef struct
23 {
24   int fd;
25   mode_t umask;
26   int exitStatus;
27   int signal;
28 } ChildParams;
29
30 static void printWaitStatus (const char *msg, int status);
31
32 /* Startup function for cloned child */
33 static int
34 childFunc (void *arg)
35 {
36   ChildParams *cp = arg;
37
```

[5] http://man7.org/tlpi/code/index.html

```
38    printf ("Child:  PID=%ld PPID=%ld\n", (long) getpid (), (long) getppid ());
39
40    /* The following changes will affect parent */
41    umask (cp->umask);
42    if (close (cp->fd) == -1)
43      {
44        perror ("(child) close");
45        exit (EXIT_FAILURE);
46      }
47
48    if (signal (cp->signal, SIG_DFL) == SIG_ERR)
49      {
50        perror ("(child) signal");
51        exit (EXIT_FAILURE);
52      }
53
54    return cp->exitStatus;        /* Child terminates now */
55  }
56
57  static void                     /* Handler for child termination signal */
58  grimReaper (int sig)
59  {
60    int savedErrno = errno;       /* In case we change 'errno' here */
61
62    /* UNSAFE: This handler uses non-async-signal-safe functions
63       (printf(), strsignal(); see Section 21.1.2) */
64    printf ("Caught signal %d (%s)\n", sig, strsignal (sig));
65    errno = savedErrno;
66  }
67
68  static void
69  usageError (char *progName)
70  {
71    fprintf (stderr, "Usage: %s [arg]\n", progName);
72  #define fpe(str) fprintf(stderr, "        " str)
73    fpe ("'arg' can contain the following letters:\n");
74    fpe ("    d - share file descriptors (CLONE_FILES)\n");
75    fpe ("    f - share file-system information (CLONE_FS)\n");
76    fpe ("    s - share signal dispositions (CLONE_SIGHAND)\n");
77    fpe ("    v - share virtual memory (CLONE_VM)\n");
78    exit (EXIT_FAILURE);
79  }
80
81  int
82  main (int argc, char *argv[])
83  {
84    char *stack;                  /* Start of stack buffer area */
85    char *stackTop;               /* End of stack buffer area */
86    int flags;                    /* Flags for cloning child */
```

```
87    ChildParams cp;               /* Passed to child function */
88    struct sigaction sa;
89    char *p;
90    int status, s;
91    pid_t pid;
92
93    printf ("Parent: PID=%ld PPID=%ld\n", (long) getpid (), (long) getppid ());
94
95    /* Set up an argument structure to be passed to cloned child, and
96       set some process attributes that will be modified by child */
97    cp.exitStatus = 22;           /* Child will exit with this status */
98
99    umask (START_UMASK);          /* Initialize umask to some value */
100   cp.umask = S_IWGRP;           /* Child sets umask to this value */
101
102   cp.fd = open ("/dev/null", O_RDWR);   /* Child will close this fd */
103   if (cp.fd == -1)
104     {
105       perror ("open");
106       exit (EXIT_FAILURE);
107     }
108
109   cp.signal = SIGTERM;          /* Child will change disposition */
110   if (signal (cp.signal, SIG_IGN) == SIG_ERR)
111     {
112       perror ("signal");
113       exit (EXIT_FAILURE);
114     }
115
116   /* Initialize clone flags using command-line argument (if supplied) */
117   flags = 0;
118   if (argc > 1)
119     for (p = argv[1]; *p != '\0'; p++)
120       {
121         if (*p == 'd')
122           flags |= CLONE_FILES;
123         else if (*p == 'f')
124           flags |= CLONE_FS;
125         else if (*p == 's')
126           flags |= CLONE_SIGHAND;
127         else if (*p == 'v')
128           flags |= CLONE_VM;
129         else
130           usageError (argv[0]);
131       }
132
133   /* Allocate stack for child */
134   stack = malloc (STACK_SIZE);
135   if (stack == NULL)
```

```
136       {
137         perror ("malloc");
138         exit (EXIT_FAILURE);
139       }
140     stackTop = stack + STACK_SIZE;        /* Assume stack grows downwards */
141
142     /* Establish handler to catch child termination signal */
143
144     if (CHILD_SIG != 0)
145       {
146         sigemptyset (&sa.sa_mask);
147         sa.sa_flags = SA_RESTART;
148         sa.sa_handler = grimReaper;
149         if (sigaction (CHILD_SIG, &sa, NULL) == -1)
150           {
151             perror ("sigaction");
152             exit (EXIT_FAILURE);
153           }
154       }
155
156     /* Create child; child commences execution in childFunc() */
157     if (clone (childFunc, stackTop, flags | CHILD_SIG, &cp) == -1)
158       {
159         perror ("clone");
160         exit (EXIT_FAILURE);
161       }
162
163     /* Parent falls through to here. Wait for child; __WCLONE option is
164        required for child notifying with signal other than SIGCHLD. */
165     pid = waitpid (-1, &status, (CHILD_SIG != SIGCHLD) ? __WCLONE : 0);
166     if (pid == -1)
167       {
168         perror ("waitpid");
169         exit (EXIT_FAILURE);
170       }
171
172     printf ("    Child PID=%ld\n", (long) pid);
173     printWaitStatus ("    Status: ", status);
174
175     /* Check whether changes made by cloned child have affected parent */
176     printf ("Parent - checking process attributes:\n");
177     if (umask (0) != START_UMASK)
178       printf ("    umask has changed\n");
179     else
180       printf ("    umask has not changed\n");
181
182     s = write (cp.fd, "Hello world\n", 12);
183     if (s == -1 && errno == EBADF)
184       printf ("    file descriptor %d has been closed\n", cp.fd);
```

```
185   else if (s == -1)
186     printf ("    write() on file descriptor %d failed (%s)\n",
187             cp.fd, strerror (errno));
188   else
189     printf ("    write() on file descriptor %d succeeded\n", cp.fd);
190
191   if (sigaction (cp.signal, NULL, &sa) == -1)
192     {
193       perror ("sigaction");
194       exit (EXIT_FAILURE);
195     }
196   if (sa.sa_handler != SIG_IGN)
197     printf ("    signal disposition has changed\n");
198   else
199     printf ("    signal disposition has not changed\n");
200
201   exit (EXIT_SUCCESS);
202 }
203
204 /* Examine a wait() status using the W* macros */
205 static void
206 printWaitStatus (const char *msg, int status)
207 {
208   if (msg != NULL)
209     printf ("%s", msg);
210
211   if (WIFEXITED (status))
212     printf ("child exited, status=%d\n", WEXITSTATUS (status));
213   else if (WIFSIGNALED (status))
214     {
215       printf ("child killed by signal %d (%s)",
216               WTERMSIG (status), strsignal (WTERMSIG (status)));
217       if (WCOREDUMP (status))
218         printf (" (core dumped)");
219       printf ("\n");
220     }
221   else if (WIFSTOPPED (status))
222     printf ("child stopped by signal %d (%s)\n",
223             WSTOPSIG (status), strsignal (WSTOPSIG (status)));
224   else if (WIFCONTINUED (status))
225     printf ("child continued\n");
226   else
227     printf ("what happened to this child? (status=%x)\n",
228             (unsigned int) status);
229 }
```

9.5 Més funcions de la família *wait*

9.5.1 La funció *waitid*

Aquesta funció proveeix funcionalitat extra i està declarada de la següent manera:

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

int waitid (idtype_t idtype, id_t id, siginfo_t *infop,
            int options);
```

Els dos primers arguments especifiquen quins fills es vol monitoritzar:

- Si `idtype` val `P_ALL`, llavors s'ignora `id` i es monitoritzen tots els fills.
- Si `idtype` val `P_PID`, llavors es monitoritza el fill amb PID `id`.
- Si `idtype` val `P_PGID`, llavors es montitoritza el fill amb amb ID de grup de processos `id`.

De totes maneres, la diferència més significativa entre `waitid` i `waitpid` és que la primera funció permet un control més precís sobre l'event que es vol monitoritzar, amb l'argument `options`:

`WEXITED` Esperar la mort d'un fill, ja sigui normal (cridant `exit` o similar) o bé "anormal" (per la recepció d'un senyal mortal).

`WSTOPPED` Esperar que un fill aturi la seva execució per la recepció d'un senyal.

`WCONTINUED` Esperar que un fill prossegueixi la seva execució perquè rep el senyal `SIGCONT`.

A més a més es poden combinar (amb l'operador `|`) els següents valors:

`WNOHANG` Té el mateix significat que per a la funció `waitpid`: si no hi ha cap fill que compleixi els criteris establerts per `idtype` i `id`, `waitid` retorna -1 en lloc de bloquejar el procés.

`WNOWAIT` Suposi's un procés pare que té un fill, i està esperant que es mori cridant la funció `wait`. El fill es mor, `wait` retorna i el pare torna a cridar `wait`. Aquesta nova crida bloquejarà el procés pare fins que un altre fill es mori, perquè el primer fill mort ja ha estat detectat. Es diu, doncs, que l'esdeveniment ha quedat consumit. El flag `WNOWAIT` fa que no es consumeixi.

La funció `waitid` retorna zero quan hi ha hagut un fill que ha complert els criteris de cerca (especificats per `idtype` i `id`, `waitid`), i omple l'estructura apuntada per `infop`. Els camps més importants d'aquesta estructura són els següents:

`si_code` Aquest camp pot tenir un dels valors següents:

`CLD_EXITED` El fill ha finalitzat la seva execució amb `_exit` o similar.

`CLD_KILLED` El fill ha mort per causa d'un senyal.

`CLD_STOPPED` El fill ha haturat la seva execució per causa d'un senyal.

`CLD_CONTINUED` El fill ha prosseguit la seva execució perquè ha rebut `SIGCONT`.

`si_pid` Identificador del procés que ha canviat d'estat (o s'ha mort, segons el cas).

`si_signo` Aquest camp sempre val `SIGCHLD`.

`si_status` Si el fill ha mort per `_exit` o similar, aquest camp conté el valor de retorn. Si ha mort per culpa d'un senyal, aquest camp conté el senyal causant de la mort del procés.

`si_uid` Aquest camp conté l'UID real del procés mort.

Quan `waitid` retorna i `options` tenia el flag `WNOHANG` activat, llavors queda el dubte de si hi ha hagut algun procés que hagi canviat d'estat. En cas negatiu, Linux posa a zero tots els camps de l'estructura apuntada per `infop`.

9.5.2 Les funcions *wait3* i *wait4*

Estan declarades de la següent manera:

```
#define _BSD_SOURCE
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait3 (int *status, int options, struct rusage *rusage);
pid_t wait4 (pid_t pid, int *status, int options,
             struct rusage *rusage);
```

- La funció `wait3` és quasi equivalent a `waitpid`, amb la diferència que dóna informació sobre els recursos utilitzats pel procés fill. La crida a `wait3` és equivalent a la següent crida a `waitpid` (excepte per l'últim argument de `wait3`):

  ```
  waitpid (-1, &status, options);
  ```

- La funció `wait4` és quasi equivalent a `waitpid`, amb la diferència que dóna informació sobre els recursos utilitzats pel procés fill. La crida a `wait4` és equivalent a la següent crida a `waitpid` (excepte per l'últim argument de `wait4`):

  ```
  waitpid (pid, &status, options);
  ```


9.5.3 El senyal *SIGCHLD*

El tema dels senyals es veurà als capítols 3 i 11, però convé fer un apunt, aquí, sobre aquest senyal i sobre com dissenyar els *signal handler* que processin aquest senyal.

La mort d'un procés és un esdeveniment asíncron, és a dir, esdevé en "qualsevol moment". Dit d'una altra manera, un procés no pot predir quan es morirà un dels seus fills (tot i que sí pot provocar-ne la mort, evidentment)[6]. Ja hem vist que el pare disposa de les funcions `wait` i similars, i per tant pot optar per dues alternatives:

- El pare pot cridar `wait` o `waitpid` (sense `WNOHANG`), per la qual cosa es queda bloquejat fins que un fill mor. Evidentment, llavors el pare no fa altra cosa que monitoritzar els seus fills.
- El pare pot, cada cert temps, de forma periòdica, cridar `waitpid` amb `WNOHANG` per tal de saber si algun dels seus fills és mort.

És obvi que seria molt més còmode que el sistema operatiu avisés un procés quan algun dels seus fills morís. Efectivament, en aquestes circumstàncies el sistema operatiu envia el senyal `SIGCHLD`, per la qual cosa es pot instal·lar un *signal handler* des d'on cridar `wait` o similars. Per defecte, els processos ignoren aquest senyal.

Ara bé, hi ha un aspecte important a tenir en compte. Suposi's que un procés rep `SIGCHDL` i executa el *signal handler* associat. Durant l'execució del *handler*, el senyal `SIGCHLD` queda bloquejat (a menys que, amb `sigaction`, s'hagi especificat que no ha de quedar bloquejat). Suposi's que, mentrestant, es mor un altre fill. Encara que es torni a generar `SIGCHLD`, el procés no el rebrà, ja que els senyals no "s'encuen". Aquest problema es pot resoldre fàcilment si, des del *signal handler*, en lloc de cridar un sol cop `wait` o similars, es crida `waitpid` de la següent manera:

```
while (waitpid (-1, NULL, WNOHANG) > 0)
  continue;
```

(si es vol saber l'estat del procés fill es pot substituir `NULL` per un punter a un enter, evidentment).

Hi ha un altre punt a tenir en compte. Suposi's que, al moment d'establir un *signal handler* per a `SIGCHLD`, ja hi ha un fill mort. Què passa llavors? En alguns UNIX de la família System V, el sistema operatiu envia el senyal `SIGCHLD`. D'altres sistemes operatius, com ara Linux, no ho fan. Novament és enormement senzill de resoldre aquest problema: només cal establir el *signal handler* abans de crear qualsevol fill.

Un últim detall és sobre la variable global `errno`. Per tal que el *signal handler* no interfereixi amb la resta del procés, pot ser bona idea guardar temporalment el valor d'aquesta variable, com es fa en l'exemple que es presenta al següent apartat.

[6] Tot i que un procés pot matar un dels seus fills, no pot predir quan exactament morirà, ja que no pot predir quan el sistema operatiu planificarà el fill per a usar la CPU i per tant rebre el senyal fatídic.

9.5.4 Exemple

L'exemple que es presenta és una adaptació del codi de les pàgines 557-558 de Kerrisk [2010]. Consisteix en un programa que rep arguments per línia de comandes. Per a cada argument es fa un fill, que s'espera un número determinat de segons abans de morir-se. Per exemple, `a.out 10 30` fa dos processos fills, el primer s'espera 10 segons i el segon 30 segons abans de morir-se. El pare es limita a monitoritzar els seus fills.

```
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>

static volatile int alive_childs;

void
print_wait_status (const char *msg, int status)
{
  if (msg)
    printf ("%s\n", msg);

  if (WIFEXITED (status))
    printf ("child exited, status = %d", WEXITSTATUS (status));
  else if (WIFSIGNALED (status))
    {
      printf ("child killed by %s (%d)", strsignal (WTERMSIG (status)),
              WTERMSIG (status));
      if (WCOREDUMP (status))
        printf (", core dumped");
    }
  else if (WIFSTOPPED (status))
    printf ("child stopped by %s (%d)", strsignal (WSTOPSIG (status)),
            WSTOPSIG (status));
  else if (WIFCONTINUED (status))
    printf ("child continued");
  else
    printf ("no idea");
  printf ("\n");
}

static void
sighandler (int signo __attribute__ ((unused)))
```

```
41 {
42   int saved_errno = errno;
43   pid_t pid;
44   int status;
45
46   while ((pid = waitpid (-1, &status, WNOHANG)) > 0)
47     {
48       print_wait_status (NULL, status);
49       alive_childs--;
50     }
51
52   if (pid == -1)
53     printf ("waitpid: %s (%d)\n", strerror (errno), errno);
54
55   errno = saved_errno;
56 }
57
58 int
59 main (int argc, char *argv[])
60 {
61   int num_childs = argc - 1;
62   if (num_childs == 0)
63     exit (0);
64   alive_childs = num_childs;
65
66   struct sigaction sa;
67   memset (&sa, 0, sizeof (sa));
68   sa.sa_flags = 0;
69   sa.sa_handler = sighandler;
70   if (sigaction (SIGCHLD, &sa, NULL) < 0)
71     {
72       printf ("sigaction: %s (%d)\n", strerror (errno), errno);
73       return -1;
74     }
75
76   /* Block SIGCHLD to prevent its delivery if a child terminates
77    * before the parent stats the sigsuspend loop below */
78
79   sigset_t set;
80   sigemptyset (&set);
81   sigaddset (&set, SIGCHLD);
82   if (sigprocmask (SIG_SETMASK, &set, NULL) < 0)
83     {
84       printf ("sigprocmask: %s (%d)\n", strerror (errno), errno);
85       return -1;
86     }
87
88   /* First, we create the children */
89
```

```
90    for (int i = 1; i < argc; i++)
91      switch (fork ())
92        {
93        case -1:
94          printf ("fork: %s (%d)\n", strerror (errno), errno);
95          exit (-1);
96
97        case 0:
98          sleep (atoi (argv[i]));
99          printf ("Child %d exiting\n", getpid ());
100         exit (0);
101
102       default:;
103       }
104
105   /* And then we wait for our children's death */
106
107   sigemptyset (&set);
108   while (alive_childs > 0)
109     if (sigsuspend (&set) < 0 && errno != EINTR)
110       {
111         printf ("sigsuspend: %s (%d)\n", strerror (errno), errno);
112         exit (-1);
113       }
114
115   printf ("All %d children have dead\n", num_childs);
116 }
```

9.6 Més sobre execució de programes

9.6.1 Execució d'scripts

Per norma general, les funcions de la família `exec` estan pensades per a executar programes compilats. Ara bé, també poden executar scripts, sempre i quan es compleixin dues condicions:

1. L'arxiu que conté l'script ha de tenir permís d'execució.
2. La primera línia de text ha de començar amb els caràcters `#!` i a continuació la ruta de l'intèrpret. Per exemple:

   ```
   #!/bin/sh
   ```

   Indica que l'script l'ha d'executar el programa `/bin/sh`.

L'apartat 27.3 (pàgines 572–574) de Kerrisk [2010] dóna detalls addicionals sobre com s'executen els scripts i com se subministra la llista d'arguments per línia de comandes.

9.6.2 Les funcions *exec* i els senyals

El procés que executa un nou programa pot tenir senyals bloquejats, senyals ignorats i *signal handlers* instal·lats. Què passa, llavors, al cridar les funcions de la família `exec`?

- Pel que fa als *signal handlers*, aquests desapareixen, ja que el codi del procés original "es perd", ja que se substitueix pel codi del nou programa que s'executa. Llavors els senyals afectats tornen al seu comportament per defecte, és a dir, com si es cridés `signal` amb `SIG_DFL` com a *signal handler*.
- El senyal `SIGCHLD` mereix certa atenció especial. Ja s'ha comentat que ignorar (explícitament, mitjançant `signal`) aquest senyal evita la creació de zombies. Ara bé, els estàndars de UNIX no diuen res al respecte. Linux deixa el senyal ignorat, per la qual cosa el nou programa no farà zombies (d'altres sistemes operatius, com ara Solaris, restableixen el comportament per defecte).
- Si algun senyal tenia un segment de pila alternatiu, es perd (flag `SA_ONSTACK` de la funció `sigaction`).
- La màscara de senyals bloquejats i el conjunt de senyals pendents d'atendre es manté.
- Els senyals ignorats segueixen sent ignorats; i els senyals que tenen el comportament per defecte el mantenen.

Capítol 10

Sistema avançat d'entrada i sortida

10.1 Introducció

L'objectiu d'aquest capítol és complementar i ampliar el material presentat al capítol 2, de manera que l'alumne disposi de noves eines i conegui aspectes importants en el funcionament del sistema d'entrada i sortida de UNIX. Concretament es presentaran eines per a treballar amb enllaços i se n'afegiran de noves per a treballar amb arxius i directoris; també es presentaran aspectes a tenir en compte relacionats amb el *buffering* i un enfocament alternatiu a l'hora de treballar amb arxius: el mapeig a memòria. Els dos últims apartats del capítol versaran sobre la funció *fcntl* — útil per a controlar alguns aspectes del funcionament d'un file descriptor — i la família de funcions *epoll*, que completen i amplien les funcionalitats de *select*.

10.2 Eines avançades d'entrada i sortida

10.2.1 Lectura i escriptura amb més d'un buffer: *readv* i *writev*

Mentre que les funcions `read` i `write` només accepten un sol buffer de memòria, les funcions següents accepten treballar amb més d'un buffer:

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

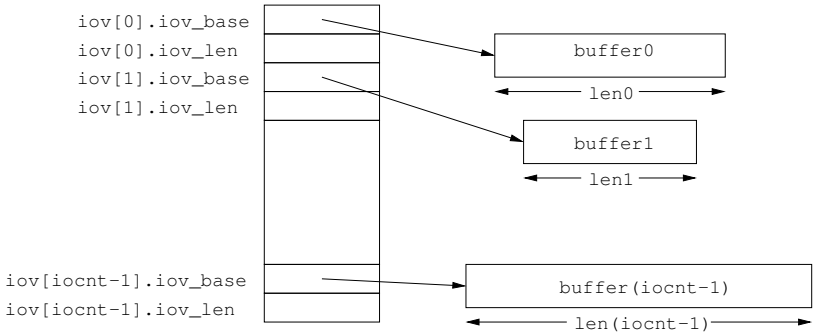

Figura 10.1: Representació gràfica de `struct iovec`

```
ssize_t readv (int fd, const struct iovec *iov, int iovcnt);
ssize_t writev (int fd, const struct iovec *iov, int iovcnt);
```

Per a aquest parell de funcions no s'indica directament el buffer ni la mida de bytes a llegir o escriure, sinó que s'empren uns *descriptors de buffer*, de tipus `struct iovec`, que indiquen quins buffers hi ha i quina mida tenen. El paràmetre `iov` és un array de descriptors, i el paràmetre `iovcnt` indica quants elements hi ha a l'array anterior. Cada element de l'array té el següent tipus:

```
struct iovec
{
  void *iov_base;
  size_t iov_len;
};
```

on `iov_base` apunta al principi del buffer, i `iov_len` indica la quantitat de bytes del buffer disponibles per a llegir o escriure. La figura 10.1 mostra gràficament un array format per elements d'aquest tipus.

- La funció `readv` llegeix, com a molt, tants bytes com indiqui la suma dels camps `iov_len` de tots els elements de l'array `iov`, i retorna la quantitat *total* de bytes llegits (no hi ha cap mena de replicació entre els buffers). Si s'han llegit menys bytes del total, pot ser que no tots els buffers s'hagin omplert, i l'últim es pot haver omplert només parcialment. A efectes de la lectura, cada buffer té la mida indicada pel corresponent camp `iov_len`, però la mida real del buffer pot ser superior (per exemple, pot contenir un byte extra per al caràcter `'\0'`).

- La funció `write` escriu, com a molt, tants bytes com indiqui la suma dels camps `iov_len` de tots els elements de l'array `iov`, i retorna la quantitat *total* de bytes escrits. Pot ser que s'hagin escrit menys bytes del total.

Quin avantatge hi ha en usar `readv` i `writev` en lloc de múltiples crides a `read` i `write`? Bàsicament dos: en primer lloc, una sola crida a `readv` o `writev` és *àtomica*, és a dir, no es barrejaran dades de diferents processos que escriguin (o llegeixin) al mateix arxiu. I en segon lloc, és més eficient una sola crida al sistema que múltiples crides.

Exemple

```
#include <sys/uio.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define STRSIZE 512

int
main (int argc, char *argv[])
{
  if (argc < 2)
    {
      fprintf (stderr, "%s: not enough arguments\n", argv[0]);
      exit (EXIT_FAILURE);
    }

  int fd;
  struct stat myStruct;
  int x;
  char str[STRSIZE] = { 0 };
  ssize_t numRead, totRequired;

  if ((fd = open (argv[1], O_RDONLY)) < 0)
    {
      perror (argv[1]);
      exit (EXIT_FAILURE);
    }

  struct iovec iov[] = {
    {&myStruct, sizeof (myStruct)},
    {&x, sizeof (x)},
    {str, sizeof (str)}
  };
  totRequired = iov[0].iov_len + iov[1].iov_len + iov[2].iov_len;

  numRead = readv (fd, iov, 3);
  if (numRead < 0)
    {
      perror ("readv");
      exit (EXIT_FAILURE);
```

```
42      }
43    if (numRead < totRequired)
44      printf ("Read fewer bytes (%ld) than requested (%ld)\n", numRead,
45              totRequired);
46    else
47      printf ("Ok\n");
48
49    close (fd);
50    return EXIT_SUCCESS;
51  }
```

10.2.2 Més funcions de lectura i escriptura

En aquest apartat es presentaran dos parells més de funcions:

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

ssize_t pread (int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite (int fd, const void *buf, size_t count, off_t
                offset);
ssize_t preadv (int fd, const struct iovec *iov, int iovcnt,
                off_t offset);
ssize_t pwritev (int fd, const struct iovec *iov, int iovcnt,
                 off_t offset);
```

Al parlar de la funció `read` s'ha donat per fet que les lectures es fan seqüencialment, és a dir, quan es llegeix d'un arxiu, la següent lectura es fa a continuació de les dades prèviament llegides. Per tant, el sistema operatiu guarda, per a cada file descriptor de cada procés, un *offset* que indica en quin punt de l'arxiu s'ha de fer la següent operació. Les funcions vistes en els dos apartats previs s'encarreguen d'actualitzar aquest *offset*. A l'apartat 2.3.3 es veurà la funció `lseek`, que permet canviar aquest *offset* sense haver de fer lectures.

Les funcions `pread` i `pwrite` funcionen exactament igual que `read` i `write`, respectivament, però reben un quart argument: l'*offset* del fitxer sobre el qual s'ha de dur a terme la lectura o escriptura. A més a més, aquestes dues funcions no actualitzen l'*offset*, que es queda tal qual estava.

Les funcions `preadv` i `pwritev` són una *combinació* de `pread` i `readv`, per una banda, i `pwrite` i `writev`, per una altra banda.

10.3 Operacions sobre arxius

10.3.1 Truncar un arxiu: *truncate* i *ftruncate*

Truncar un arxiu significa modificar-ne la seva mida sense necessitat de fer lectures ni escriptures. Per a tal fi hi ha dues funcions:

```
#include <unistd.h>

int truncate (const char *pathname, off_t length);
int ftruncate (int fd, off_t length);
```

El paràmetre `length` indica la nova mida de l'arxiu; l'única diferència entre aquestes dues funcions és com s'especifica l'arxiu a truncar:

- La funció `truncate` rep un nom d'arxiu, ja sigui una ruta absoluta i relativa.
- La funció `ftruncate` rep un file descriptor; per tant, l'arxiu ja ha d'estar prèviament obert.

Si la nova mida de l'arxiu és inferior a l'anterior, les dades *sobrants* es descarten. Si la nova mida és més gran, l'espai que no existia es crea a partir de zeros (és a dir, `'\0'`).

10.3.2 Atributs d'un arxiu: *stat*, *fstat* i *lstat*

Un arxiu no només conté dades, sinó també una sèrie d'atributs (generalment coneguts sota el nom de *metadades*) com ara el nom de l'arxiu, la seva mida, el seu propietari, la data i l'hora de creació, els permisos d'accés, etc. Hi ha tres funcions que permeten obtenir aquests atributs:

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

int stat (const char *path, struct stat *buf);
int lstat (const char *path, struct stat *buf);
int fstat (int fd, struct stat *buf);
```

- La funció `stat` obté els atributs de l'arxiu indicat per `path`, seguint els enllaços simbòlics (més informació a l'apartat 2.4.1 sobre els enllaços simbòlics).
- La funció `lstat` obté els atributs de l'arxiu indicat per `path`, sense seguir els enllaços simbòlics. Si l'arxiu és un enllaç simbòlic, llavors s'obtenen els atributs del propi enllaç simbòlic.
- La funció `fstat` obté els atributs de l'arxiu, prèviament obert, indicat pel file descriptor `fd`.

El paràmetre `buf` apunta a una estructura que conté diversos camps, que representen els diferents atributs de l'arxiu. No es descriuran tots els camps que formen l'estructura, sinó només els que es consideren més rellevants.

`st_dev` Identificador del dispositiu on resideix l'arxiu.

Macro	Tipus d'arxiu
`S_ISREG()`	Arxiu regular
`S_ISDIR()`	Directori
`S_ISCHR()`	Dispositiu de caràcters
`S_ISBLK()`	Dispositiu de blocs
`S_ISFIFO()`	FIFO
`S_ISSOCK()`	Socket de UNIX
`S_ISLNK()`	Enllaç simbòlic

Taula 10.1: Taula de macros de tipus d'arxiu per a `st_mode`

`st_ino` Número d'inode de l'arxiu. El número d'inode és l'identificador únic de l'arxiu (dins d'una partició o dispositiu), ja que un arxiu pot tenir diversos noms (vegi's l'apartat 2.4.1 sobre el concepte d'enllaç dur). La combinació de `st_dev` i `st_ino` identifiquen unívocament l'arxiu.

`st_mode` Combinació de tipus d'arxiu i permisos d'accés. El tipus d'arxiu indica si es tracta d'un arxiu regular, d'un directori, d'un enllaç simbòlic, d'un arxiu de dispositiu, etc. Els permisos d'accés indiquen els tipus d'operacions que els diferents usuaris poden realitzar sobre l'arxiu. A l'apartat 10.3.3 es parlarà en detall sobre els permisos d'un arxiu, i a l'apartat 10.5.5 sobre els tipus d'arxiu.

`st_nlink` Quantitat d'enllaços durs que té el fitxer.

`st_uid` Identificador (numèric) de l'usuari propietari de l'arxiu. Cal recordar que el sistema UNIX és multi-usuari, i identifica internament cada usuari amb un identificador numèric.

`st_gid` Identificador (numèric) del grup propietari de l'arxiu. A més dels usuaris, hi ha també grups, cadascun dels quals té un nom i un identificador numèric.

`st_size` Mida (en bytes) de l'arxiu (sense incloure les metadades).

Pel que fa al camp `st_mode`, hi ha una sèrie de macros que permeten determinar fàcilment el tipus d'arxiu, com es pot veure a la taula 10.1.

La macro `S_ISLNK` només retornarà cert si s'ha cridat `lstat`, ja que `stat` sempre segueix els enllaços simbòlics. La macro `S_ISSOCK` només està disponible si s'ha definit la macro `_BSD_SOURCE` abans d'incloure `sys/stat.h`. Pel que fa als permisos, se'n parlarà amb més detall a l'apartat 10.3.3.

A l'apartat 10.6 es veurà un exemple on es cridarà la funció `stat` per tal d'obtenir la mida d'un arxiu.

10.3.3 Permisos d'un arxiu: *umask*, *chmod* i *fchmod*

Abans d'entrar en matèria sobre les funcions que permeten canviar els permisos d'un arxiu, cal donar algunes definicions:

- El *propietari* d'un arxiu és l'usuari que ha creat l'arxiu, tot i que l'administrador (l'usuari `root`, amb identificador zero) pot canviar el propietari de qualsevol arxiu.
- El grup propietari és, per defecte, el grup a què pertany l'usuari propietari de l'arxiu, tot i que un usuari pot pertànyer a diferents grups.

A l'hora d'assignar permisos, s'assignen permisos a:

- L'usuari propietari de l'arxiu.
- Els usuaris que pertanyen al grup propietari de l'arxiu.
- Tota la resta d'usuaris.

Els permisos que es poden atorgar són tres:

- Lectura: permet llegir l'arxiu i, en un directori, mostrar els arxius que conté (només el nom).
- Escriptura: permet escriure a l'arxiu i, en un directori, crear i eliminar-ne directoris, encara que no es tingui cap permís sobre els arxius que es vol eliminar. És a dir, si un usuari té permís d'escriptura sobre un directori, pot eliminar arxius d'altres usuaris en aquest directori.
- Execució. Per als arxius regulars, el permís d'execució indica que l'arxiu és un programa o un *script*, i que es pot executar. Per als directoris, vol dir que el directori es pot *atravessar*, és a dir, llistar-ne el contingut i accedir als seus directoris. Si es té permís d'execució però no de lectura sobre un directori, es pot accedir als seus arxius si se'n sap el nom. Per tal d'afegir o eliminar arxius en un directori, cal tenir permís d'escriptura i d'execució sobre el directori.

A més d'aquests permisos, un arxiu pot tenir activats tres permisos addicionals:

- Permís `SETUID`: només s'aplica als executables (o *scripts*, i indica que el programa tindrà els permisos de l'usuari propietari del programa, en lloc de tenir els permisos de l'usuari que executa el programa.
- Permís `SETGID`: només s'aplica als executables (o *scripts*, i indica que el programa tindrà els permisos del grup propietari del programa.
- *Sticky bit*: actualment aquest permís només té sentit en els directoris. S'ha comentat anteriorment que si un usuari té permís d'escriptura sobre un directori, pot crear i eliminar arxius, fins i tot si pertanyen a d'altres usuaris. Si l'*sticky bit* està activat, un usuari només podrà eliminar arxius del directori si és el propietari del directori o bé si té permís d'escriptura al directori i és el propietari de l'arxiu que vol eliminar.

Constant	Valor octal	Permís
S_ISUID	04000	SETUID
S_ISGID	02000	SETGID
S_ISVTX	01000	*Sticky bit*
S_IRUSR	0400	Lectura per al propietari
S_IWUSR	0200	Escriptura per al propietari
S_IXUSR	0100	Execució per al propietari
S_IRGRP	040	Lectura per al grup
S_IWGRP	020	Escriptura per al grup
S_IXGRP	010	Execució per al grup
S_IROTH	04	Lectura per als altres
S_IWOTH	02	Escriptura per als altres
S_IXOTH	01	Execució per als altres

Taula 10.2: Taula de constants de permisos per a `st_mode`

Aplicat als directoris, els bits `SETUID` i `SETGID` tenen d'altres usos; per a més detalls, es recomana consultar [Kerrisk, 2010, capítols 8 i 9]).

Al parlar de la funció `stat` i similars (apartat 10.3.2) s'ha parlat del camp `st_mode`, que combina el tipus d'arxiu amb els seus permisos. Hi ha una sèrie de constants que permeten comprovar si un determinat permís està activat (per exemple, si es fa una *and* binària de la constant `S_ISUID` amb `st_mode` es pot saber si el permís `SETUID` està activat per a un arxiu). Aquestes constants són les de la taula 10.2.

Fins ara s'ha presentat l'esquema tradicional de permisos del sistema UNIX. Des de fa cert temps existeix el que es coneix amb el nom de *llistes de control d'accés* (ACL), que permeten especificar permisos per a usuaris i/o grups concrets. En aquest capítol no es parlarà de les llistes de control d'accés (per a més informació es pot consultar [Kerrisk, 2010, capítol 17]). També cal tenir en compte que alguns sistemes de fitxers, com ara *ext2*, suporten alguns flags específics que poden limitar els permisos assignats a un arxiu (per a més informació es pot consultar [Kerrisk, 2010, apartat 15.5]).

Un cop explicada la teoria, ja es pot passar a veure les funcions relacionades amb els permisos.

Funció *umask* La funció `umask` determina la màscara de permisos, que són els permisos que sempre s'han de desactivar quan es crea un arxiu o directori. Per exemple, si la màscara val 022 i es vol crear un arxiu amb permisos 666, a la pràctica els permisos que tindrà l'arxiu seran 644 ($666 - 022$).

La funció `umask` permet canviar la màscara de permisos:

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

mode_t umask (mode_t mask);
```


La nova màscara es pot especificar com un valor numèric o bé combinant (amb l'operador *or* binària) les constants de la taula 10.2. La funció retorna l'antiga màscara de permisos.

Funcions *chmod* i *fchmod* Aquestes dues funcions permeten canviar els permisos d'un arxiu:

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

int chmod (const char *path, mode_t mode);
int fchmod (int fd, mode_t mode);
```

La diferència entre aquestes dues funcions és com s'especifica l'arxiu del qual es volen canviar els permisos. Si `path` és un enllaç simbòlic, es canvien els permisos del fitxer referenciat, no els de l'enllaç simbòlic (els enllaços simbòlics sempre tenen permisos 777, no es poden canviar i s'ignoren totalment). Els permisos es poden especificar com un valor numèric o bé combinant (amb l'operador *or* binària) les constants de la taula 10.2.

10.3.4 Propietari d'un arxiu: *chown*, *lchown* i *fchown*

Com que UNIX és un sistema multi-usuari, cada arxiu té associat un propietari (usuari propietari i grup propietari), que conjuntament amb els permisos (explicats a l'apartat 10.3.3) defineixen quins usuaris poden accedir a quins arxius. Quan un arxiu es crea, se li assigna un usuari propietari i un grup propietari; si el lector està interessat en conèixer les regles concretes que segueix el sistema operatiu per assignar usuari i grup propietari, pot consultar [Kerrisk, 2010, apartat 15.3.1].

Les funcions següents permeten canviar el propietari d'un arxiu:

```
#include <unistd.h>

int chown (const char *path, uid_t owner, gid_t group);
int lchown (const char *path, uid_t owner, gid_t group);
int fchown (int fd, uid_t owner, gid_t group);
```

La diferència entre aquestes tres funcions és equivalent a la diferència entre `stat`, `lstat` i `fstat`: `chown` segueix els enllaços simbòlics, `lchown` no els segueix i `fchown` actua sobre un arxiu prèviament obert.

- El paràmetre `owner` indica l'identificador numèric del nou propietari de l'arxiu, o bé -1 si no se'n vol canviar el propietari. Cal ser el superusuari per a canviar el propietari d'un arxiu.
- El paràmetre `group` indica l'identificador numèric del nou grup propietari de l'arxiu, o bé -1 si no se'n vol canviar el grup propietari. El superusuari pot

especificar qualsevol valor de grup; els altres usuaris només poden especificar un grup del qual en siguin membres.

Quan es canvia l'usuari propietari o el grup propietari d'un arxiu, es desactiven automàticaments els bits `SETUID` i `SETGID`. Ara bé, si el grup té desactivat el permís d'execució o bé si `path` es refereix a un directori, llavors el bit `SETGID` no es desactiva automàticament (per a més detalls, es pot consultar [Kerrisk, 2010, pàgina 293]).

10.3.5 Canviar el nom d'un arxiu: *rename*

Per tal de canviar el nom d'un arxiu o directori (també es pot emprar per a moure un arxiu), es disposa de la funció `rename`:

```
#include <stdio.h>

int rename (const char *oldpath, const char *newpath);
```

El paràmetre `oldpath` és un arxiu o directori existeix, que canvia el seu nom per l'indicat a `newpath`. Cal tenir en compte que `rename` tan sols modifica entrades de directori, és a dir, no fa còpies ni moviments del contingut dels arxius implicats.

La funció `rename` segueix les normes següents:

1. Si `newpath` ja existeix, se sobreesciu.
2. Si `newpath` i `oldpath` es refereixen al mateix arxiu, no es fa cap mena de canvi.
3. No se segueixen els enllaços simbòlics, ni a `oldpath` ni a `newpath`.
4. Si `oldpath` no és un directori, llavors `newpath` no pot ser un directori. Si el que es vol és moure un arxiu a un altre directori alhora que se li canvia el nom, cal especificar els directoris involucrats; per exemple:

   ```
   rename ("sub1/x", "sub2/y");
   ```

   mou l'arxiu `x` del directori `sub1` i el col·loca al directori `sub2`, passant-se a dir `y`.
5. Si `oldpath` és un directori, llavors `newpath` ha de ser el nom d'un directori buit o que no existeixi. Si `oldpath` és el nom d'un arxiu existent o d'un directori no buit, llavors `rename` falla.

   A més a més, `newpath` no pot contenir el mateix directori `oldpath`; és a dir, la següent crida és errònia:

   ```
   rename ("/home/mtk", "/home/mtk/bin");
   ```
6. Finalment, els arxius referenciats per `oldpath` i `newpath` han de pertànyer a la mateixa partició.

10.4 Buffering sobre arxius

10.4.1 Introducció

A l'apartat 2.2 s'han vist una sèrie de funcions per a llegir i escriure amb file descriptors. Aparentment, quan es crida `write` per a escriure a un arxiu, les dades realment s'escriuen a l'arxiu, o almenys això és el que s'espera que passi. En realitat, però, el sistema operatiu no ho fa realment d'aquesta manera. Per motius d'eficiència i de rendiment, hi ha una sèrie de *buffers* que actuen com una *cache* de disc. Quan es crida `write` per a escriure a arxiu, en realitat s'escriuen a un *buffer* del sistema operatiu; si després un altre procés llegeix de l'arxiu, el sistema operatiu comprova si les dades demanades estan en algun dels seus *buffers* (del sistema operatiu, no del procés), d'aquesta manera es poden accelerar les operacions.

No és l'objectiu d'aquest apartat descriure detalladament com funciona el *buffering* de UNIX, ni de quines són les millors estratègies per tal d'accelerar al màxim el rendiment, sinó simplement conèixer bàsicament com funciona i de quins mecanismes disposa el programador per tal d'alterar-ne el comportament.

10.4.2 *Buffering* en la llibreria de C

Les funcions de la llibreria de C per a treballar amb arxius, com ara `fprintf`, `fscanf`, `fread` i `fwrite` (i també `printf` i `scanf`) tenen el seu propi mecanisme de *buffering*, a part del del propi sistema operatiu. L'avantatge d'aquest buffer extra és que estalvia crides al sistema, que comporten un canvi de context que té un cost més elevat que el de cridar una funció.

La llibreria de C proveeix tres tipus diferents de *buffering*:

Buffering complet Aquesta mena de buffering s'empra normalment per als arxius que estan a disc. Només es crida `read` quan el buffer està buit, i només es crida `write` quan el buffer està ple. Si l'aplicació finalitza abans que s'hagin escrit les dades pendents al buffer, aquestes dades es perden. La funció `exit` traspassa els buffers a arxiu abans de finalitzar el procés, però la funció `_exit` no ho fa.

Buffering de línia Aquesta mena de bufferin s'empra normalment quan "l'arxiu" es refereix a un terminal, és a dir, a pantalla o a teclat. En aquest cas, cada cop que es llegeix o s'escriu una línia es crida `read` o `write`. Pot ser que el buffer no sigui prou gran per a encabre-hi una línia; en aquest cas, no s'esperarà a escriure o llegir una línia per tal de cridar `read` o `write`.

Sense buffering La llibreria de C no posa cap buffer.

Quan hi ha buffer, la funció `fflush` obliga la llibreria de C a escriure les dades que hi hagi pendents al buffer. La funció `setvbuf` permet canviar la disposició del buffer per a un determinat arxiu (representat per un `FILE *`):

```
#include <stdio.h>

int setvbuf (FILE *stream, char *buf, int mode, size_t size);
```

- El paràmetre `buf` apunta a una zona de memòria que servirà de buffer *permanent* (almenys mentre no es torni a canviar) per a les funcions de la llibreria de C. Per tant, ha de ser estàtic (una variable global) o ubicat amb `malloc` (o similar). Sota cap concepte s'ha d'utilitzar una variable automàtica (local que no s'hagi declarat amb `static`) com a buffer per a la funció `setvbuf`. El paràmetre `size` indica la mida, en bytes, d'aquest buffer.
- Si al paràmetre `buf` se li passa un punter `NULL`, llavors la pròpia llibreria de C ubica un buffer (amb una excepció, que es veurà a continuació). La llibreria de C de Linux no permet que el paràmetre `size` indiqui la mida del buffer que ha d'ubicar la llibreria de C, per tant si `buf` val `NULL` s'ignora el valor de `size`.
- El paràmetre `mode` indica quin tipus de buffer s'ha d'emprar per a `stream`:

  `_IONBF` Sense buffer. Cada crida a una funció de la llibreria de C que llegeixi o escrigui correspondrà a una crida a `read` o `write`. Els arguments `buf` i `size` s'ignoren completament, i poden valdre `NULL` i 0, respectivament.

  `_IOLBF` *Buffering* de línia.

  `_IOFBF` *Buffering* complet.

La funció `setvbuf` retorna zero si no hi ha hagut cap error. A part de la funció `setvbuf`, hi ha dues funcions més:

```
#include <stdio.h>

void setbuf (FILE *stream, char *buf);

#define _BSD_SOURCE
#include <stdio.h>

void setbuffer (FILE *stream, char *buf, size_t size);
```

- La crida a la funció `setbuf` és equivalent a la següent crida a `setvbuf`:

  ```
  setvbuf (stream, buf, (buf != NULL) ? _IOFBF : _IONBF,
           BUFSIZ);
  ```

  per a un valor de `BUFSIZ` típic de 8 KB.

- La crida a la funció `setbuffer` és equivalent a la següent crida a `setvbuf`:

  ```
  setvbuf (stream, buf, (buf != NULL) ? _IOFBF : _IONBF,
           size);
  ```


La funció *fflush* Independentment del mode de buffer que estigui actiu per a un `FILE *`, es pot forçar la llibreria de C a *buidar* el buffer, de manera que les dades pendents d'escriptura es passin al buffer del sistema operatiu cridant la funció `write`. A tal fi hi ha la funció següent:

```
#include <stdio.h>

int fflush (FILE *stream);
```

on `stream` és l'arxiu pel qual es vol buidar el buffer. Si es passa un punter `NULL`, llavors es buiden els buffers de tots els `FILE *` oberts pel procés. En moltes llibreries de C, incloent la de Linux, si `stdin` i `stdout` es refereixen a un terminal, llavors cada cop que es llegeix de `stdin` es fa prèviament un buidat del buffer de `stdout`.

10.4.3 *Buffering* del sistema operatiu

En l'apartat anterior s'ha explicat el *buffering* de la llibreria de C, en aquest apartat s'explica el que aplica el propi sistema operatiu. Tal com s'ha explicat, per defecte totes les operacions d'entrada i sortida de UNIX fan servir un buffer posat pel propi sistema operatiu. Hi ha una sèrie de funcions per a forçar l'escriptura a disc de les dades presents als buffers, per forçar que les operacions siguin síncrones (més endavant es veurà què s'entén per *operació síncrona*) i fins i tot com es pot obligar un arxiu a saltar-se els buffers del sistema operatiu.

Funcions *fsync*, *fdatasync* i *sync* La funció `fsync` fa que totes les dades d'un arxiu (i les seves *metadades*, com ara el nom, la mida, el propietari, la data de creació, etc) presents als buffers s'escriguin realment a disc. Un cop la funció retorna, es pot estar segur que la versió de l'arxiu a disc reflecteix totes les escriptures que s'hi han fet. La funció `fdatasync`, en canvi, només escriu les dades de l'arxiu, no actualitza les metadades.

```
#include <unistd.h>

int fsync (int fd);
int fdatasync (int fd);
```

Cal tenir en compte que els discs durs actuals tenen la seva pròpia *cache*; per tant, quan qualsevol de les dues funcions anteriors retorna, pot ser que les dades de l'arxiu no estiguin físicament a disc, sinó a la seva *cache*. La comanda `hdparm -W0` serveix per a desactivar aquesta *cache*, a costa de reduir molt el rendiment del disc.

La funció següent causa que s'escriguin a disc tots els buffers del sistema operatiu corresponent a tots els arxius oberts:

```
#include <unistd.h>

void sync (void);
```

En Linux, `sync` retorna després d'haver escrit a disc tots els buffers, tot i que pot ser que, en d'altres UNIX, la funció retorni abans d'haver transferit a disc els buffers.

Flags *O_SYNC* i *O_FSYNC* A l'apartat 2.2.3 s'ha vist un *flag* anomenat `O_SYNC`, que es pot emprar a l'hora d'obrir un arxiu. La seva finalitat és forçar que les escriptures siguin síncrones, és a dir, després de cridar qualsevol funció que escrigui dades (com ara `write` o les seves variants) s'escriuen a disc totes les dades dels buffers que contene dades de l'arxiu, i també s'actualitzen les metadades. Si es vol que la *sincronia* afecti únicament les dades i no les metadades, llavors en lloc d'especificar el flag `O_SYNC` s'especifica `O_DSYNC`. Cal tenir en compte que l'escriptura síncrona afecta força negativament el rendiment de les escriptures.

Flag *O_DIRECT* Queda pendent un únic aspecte relacionat amb els buffers del sistema operatiu: el flag `O_DIRECT`. En els dos flags explicats anteriorment, els buffers del sistema operatiu segueixen en funcionament, tant per lectura com per escriptura, amb l'única diferència que cada escriptura força que s'actualitzi l'arxiu a disc amb les dades dels buffers. El flag `O_DIRECT` simplement *desactiva* els buffers per a un determinat arxiu, la qual cosa encara penalita més el rendiment de l'arxiu. Llavors, quin és el sentit de `O_DIRECT`? Està pensat per a aplicacions que fan servir els seus propis mecanismes de *buffering* (com ara alguns sistemes gestors de bases de dades), i per tant no es desitja que el sistema operatiu faci una feina que ja fa la pròpia aplicació.

A part de la penalització en el rendiment, hi ha une restriccions extra:

1. El buffer que es proveeix a les funcions `read` o `write` (o similars) ha d'estar *alineat* a la mida del bloc, és a dir, l'adreça del buffer ha de ser un múltiple de la mida de bloc (512 bytes).

2. L'*offset* de l'arxiu, a l'hora de fer la lectura o l'escriptura, ha de ser un múltiple de la mida de bloc.

3. La mida de les dades que es transfereixen ha de ser un múltiple de la mida de bloc.

10.5 Enllaços i directoris

10.5.1 Creació i eliminació d'enllaços: *link* i *unlink*

Per a crear i eliminar enllaços hi ha les tres funcions següents:

```
#include <unistd.h>

int link (const char *oldpath, const char *newpath);
int unlink (const char *path);
```


- La funció `link` crea un nou enllaç dur per a un arxiu ja existent, de nom `oldpath`. El nom del nou enllaç dur és `newpath`. És a dir, que `oldpath` es refereix a un arxiu que ja existeix, que a partir d'ara tindrà un nom addicional: `newpath`.

  En Linux i Solaris, si `oldpath` és un enllaç simbòlic, es crea un enllaç dur a l'enllaç simbòlic — és a dir, `link` no segueix els enllaços simbòlics. En d'altres implementacions de UNIX sí segueix els enllaços simbòlics.

- La funció `unlink` fa tot el contrari de `link`, és a dir, elimina l'enllaç dur. Quan l'últim enllaç dur de l'arxiu s'ha eliminat, llavors s'elimina completament l'arxiu. De totes maneres, cal tenir en compte que si l'arxiu està obert, no s'eliminarà fins que no es tanquin tots els file descriptors oberts. Ara bé, `unlink` no permet eliminar directoris. La funció `unlink` no segueix els enllaços simbòlics.

10.5.2 Treballar amb enllaços simbòlics: *symlink* i *readlink*

La funció `link` permet crear enllaços durs; per a la creació d'enllaços simbòlics es disposa de la funció `symlink`:

```
int symlink (const char *filepath, const char *linkpath);
```

Val la pena notar que la funció `symlink` no exigeix que `filepath` existeixi — es pot crear un enllaç simbòlic a un arxiu que no existeix. A vegades pot interessar saber a quin arxiu apunta un determinat enllaç simbòlic. Per a tal propòsit es disposa de la funció `readlink`:

```
#include <unistd.h>

ssize_t readlink (const char *path, char *buffer, size_t bufsiz);
```

La funció retorna la quantitat de bytes que s'han escrit a la zona de memòria apuntada per `path`. Aquesta funció té un inconvenient, i és que se li ha de proporcionar un buffer de memòria suficientment gran per tal que hi pugui cabre l'arxiu apuntat per l'enllaç simbòlic (que es passa com a argument al paràmetre `path`). La mida d'aquest buffer s'indica amb el paràmetre `bufsiz`. Alternativament es pot fer que aquest buffer tingui una mida de `PATH_MAX` bytes. Aquesta constant està definida a `limits.h`.

Un altre inconvenient és que `readlink` no posa un caràcter `'\0'` al final del nom de l'arxiu apuntat.

10.5.3 Creació i eliminació de directoris: *mkdir* i *rmdir*

Per tal de crear un directori, es disposa de la funció següent:

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

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

Per tal que la crida a `mkdir` no falli, `path` no ha de correspondre a cap arxiu ja existent, i han d'existir els directoris dels nivells superiors; és a dir, per tal que es pugui crear el directori `aaa/bbb/ccc`, ja han d'existir tant `aaa` com `aaa/bbb`.

El paràmetre `mode` indica els permisos de l'arxiu, que s'especifiquen de la mateixa manera que en les funcions `open` i `creat`. [Kerrisk, 2010, apartat 18.6] dóna detalls addicionals sobre com s'estableixen els permisos per als directoris.

Per a eliminar directoris es disposa de la següent funció:

```
#include <unistd.h>

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

Per tal que `rmdir` pugui eliminar el directori, cal que estigui buit. Aquesta funció segueix els enllaços simbòlics excepte en *l'últim pas*; per exemple:

```
rmdir ("/home/user/dir");
```

tant `/home` com `/home/user` poden ser enllaços simbòlics, però no ho pot ser `/home/user/dir`.

10.5.4 Funció *realpath*

La funció `realpath` dóna la ruta absoluta corresponent a qualsevol ruta relativa (o absoluta en què hi hagi enllaços simbòlics, referències a `.` `..`, etc), de manera que s'obté una ruta absoluta sense enllaços simbòlics:

```
#include <stdlib.h>

char *realpath (const char *path, char *resolved);
```

La funció `realpath` resol la ruta indicada per `path` i la ruta resolta s'escriu a la zona de memòria apuntada per `resolved`. En Linux, si es passa un punter `NULL` a `resolved`, llavors la funció s'encarrega de demanar la memòria necessària, l'adreça de la qual és el valor de retorn (el programador és responsable d'alliberar la memòria que `realpath` ha demanat).

10.5.5 Tipus d'arxius

Fins al moment s'han vist tres tipus d'arxius:

- Arxius *regulars*, que són els arxius que contenen dades com ara programes, documents, imatges, etc. La immensa majoria dels arxius d'un disc són arxius regulars.

- Directoris: tot aquest apartat ha versat sobre ells.
- Enllaços simbòlics: se n'ha parlat a l'apartat 2.4.1.

A més d'aquests tipus, n'hi ha uns quants més:

- Dispositiu de caràcter: indica punts d'accés a dispositius que s'hi accedeix byte a byte (com ara un terminal). Aquesta mena d'arxius acostumen a residir al directori `/dev`.
- Dispositiu de bloc: indica punts d'accés a dispositius que s'hi accedeix per blocs (com ara un disc dur). Aquesta mena d'arxius acostumen a residir al directori `/dev`.
- Pipes: al capítol de mecanismes de comunicació entre processos es veurà un mecanisme anomenat *fifo* o *named pipe*; aquesta mena de mecanismes creen uns arxius d'aquest tipus.
- Sockets: al capítol de mecanismes de comunicació entre processos es veurà un mecanisme anomenat *sockets de domini local*; aquesta mena de mecanismes creen uns arxius d'aquest tipus.

10.5.6 Concepte de directori arrel

El directori arrel d'un procés indica a partir d'on comencen les rutes absolutes; en la immensa majoria de casos, el directori arrel d'un procés coincideix amb el directori arrel real d'un sistema. Els processos que s'executen amb permisos de superusuari poden canviar el directori arrel amb la següent funció:

```
#define _BSD_SOURCE
#include <unistd.h>

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

Canviar el directori arrel d'un procés sol ser una mesura de segureat, ja que confina el procés a una àrea específica del sistema d'arxius. Un exemple típic és el programa `ftp`, en què es crea una zona reservada exclusivament per als usuaris que hi entren de forma anònima.

10.6 Mapeig d'arxius a memòria

Tot aquest capítol, fins al moment, ha versat sobre l'ús de file descriptors per a realitzar operacions d'entrada i sortida, i concretament sobre com treballar amb arxius i directoris. L'objectiu d'aquest apartat és presentar un enfocament alternatiu a les operacions sobre arxius: el mapeig d'arxius a memòria, servint-se de la funció `mmap`. Addicionalment, també s'explicarà com emprar aquesta funció `mmap` per a demanar memòria directament al sistema operatiu.

10.6.1 Introducció

La funció `mmap` serveix per a mapejar arxius o demanar memòria:

- Mapeig d'arxiu: d'aquesta manera s'aconsegueix que el contingut d'un arxiu passi a residir a la memòria, i el procés pot accedir al contingut de l'arxiu de la mateixa manera que accedeix a memòria. Fins i tot pot modificar el contingut de l'arxiu simplement modificant el contingut de la memòria. El sistema operatiu carrega el contingut de l'arxiu a memòria a mesura que el procés ho va demanant — és a dir, si a una part de l'arxiu no s'hi accedeix mai, és probable que el sistema operatiu no la llegeixi i no la carregui a memòria.

- Demanar memòria: llavors la funció `mmap` actua d'una forma similar a `malloc` (de fet, certes implementacions de `malloc` se serveixen de la funció `mmap` per a obtenir la memòria). La funció `mmap` inicialitza la memòria obtinguda a zero. D'aquest tipus de mapeig també se'n diu *mapeig anònim*.

A més a més, `mmap` permet compartir la memòria amb d'altres processos:

- Mapeig privat (`MAP_PRIVATE`): els canvis a la zona de memòria són privats per al procés que els efectua. Si es tracta d'un arxiu, aquest no queda modificat pels canvis que es fan a la zona de memòria; si es tracta d'un mapeig *anònim*, la zona de memòria no es comparteix amb d'altres processos.

- Mapeig compartit (`MAP_SHARED`): si es tracta d'un arxiu, aquest queda modificat pels canvis que es fan a la zona de memòria (i si d'altres processos tenen mapejat el mateix arxiu, llavors també veuen els canvis); si es tracta d'un mapeig *anònim*, llavors la zona de memòria és compartida entre diferents processos (entre pare i fills).

10.6.2 Les funcions *mmap* i *munmap*

Aquestes dues funcions estan declarades de la següent manera:

```
#include <sys/mman.h>

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

- Per a la funció `mmap`, el paràmetre `addr` indica l'adreça a on s'ha de crear el mapeig de memòria. Si es passa un punter `NULL`, el sistema operatiu decideix l'adreça (que és la millor manera de fer-ho). Per a la funció `munmap`, `addr` indica l'adreça d'un mapeig.

Constant	Descripció
`PROT_NONE`	La regió no és accessible
`PROT_READ`	La regió permet lectures
`PROT_WRITE`	La regió permet escriptures
`PROT_EXEC`	La regió conté codi executable

Taula 10.3: Valors vàlids per al paràmetre `prot`

- Per a la funció `mmap`, el paràmetre `length` indica la mida del mapeig a memòria. Si es mapeja tot un arxiu, el més lògic és que `length` sigui justament la mida de l'arxiu; tot i això, és legítim mapejar només una part d'un arxiu. Si es tracta d'un mapeig anònim, `length` és a quantitat de memòria que es demana. Per a la funció `munmap`, `length` és la quantitat de memòria del mapeig apuntat per `addr`.

- El paràmetre `prot` indica quines operacions es podran dur a terme a la regió de memòria obtinguda, segons la taula 10.3. Les constants d'aquesta taula es poden combinar amb l'operador `|` (*or* binària).

- El paràmetre `flags` indica, entre d'altres aspectes, si es tracta d'un mapeig privat (`MAP_PRIVATE`) o compartit (`MAP_SHARED`). S'ha d'incloure una i només una d'aquestes dues constants com a argument a `flags`. Si es tracta d'un mapeig anònim, hi ha la constant `MAP_ANONYMOUS`, que s'ha de combinar amb una (i només una) de les dues constants anteriors. Hi ha d'altres constants, que es poden consultar a la pàgina manual de `mmap`.

- Els paràmetres `fd` i `offset` tenen sentit únicament quan es mapegen arxius. En aquest cas, `fd` és el file descriptor d'un arxiu prèviament obert (per la qual cosa cal obrir un arxiu abans de poder-lo mapejar), i `offset` és la posició de l'arxiu a la qual comença el mapeig (és a dir, es pot mapejar un tros de l'arxiu que no comenci, necessàriament, pel principi). En un mapeig anònim, `fd` ha de valdre -1 i el paràmetre `offset` s'ignora. Pot ser que el sistema operatiu posi restriccions d'alineament sobre el paràmetre `offset` (per exemple, que hagi de ser un múltiple de la mida de la pàgina), convé consultar la pàgina manual de `mmap`.

Les proteccions `PROT_NONE` i `PROT_EXEC` requereixen una certa explicació. La primera d'elles pot servir per a crear zones de *protecció*. Suposi's que es vol verificar que un programa no accedeix fora d'una determinada zona de memòria. Es pot crear un mapeig abans de la zona de memòria i un altre mapeig després amb la protecció `PROT_NONE`, de forma que si el procés intenta sortir-se dels límits de la zona de memòria a la qual es vol que pugui tenir accés, el sistema operatiu enviarà el senyal `SIGSEGV` i d'aquesta manera se sabrà que hi ha un error a l'aplicació. La

segona, PROT_EXEC, té utilitat per a programes *loader*, és a dir, que serveixen per a permetre l'execució d'altres programes.

Exemple L'exemple següent consisteix en l'ús de la funció mmap per a mapejar un arxiu a memòria (el mapeig és privat). L'aplicació obre un arxiu i compta quants caràcters hi ha que siguin dígits.

```
1  #include <sys/types.h>
2  #include <sys/stat.h>
3  #include <sys/mman.h>
4  #include <ctype.h>
5  #include <fcntl.h>
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <unistd.h>
9
10 int
11 main (int argc, char *argv[])
12 {
13   int fd;         // file descriptor
14   char *buff;     // punter a on hi haur  l'arxiu mapejat
15   struct stat sb; // per a saber la mida de l'arxiu
16   int counter;    // comptador de car cters llegits
17   int digits;     // comptador de d gits
18
19   if (argc < 2)
20     {
21       fprintf (stderr, "Falten arguments.\n");
22       return -1;
23     }
24   // abans d'executar mmap hem d'obrir l'arxiu
25   if ((fd = open (argv[1], O_RDONLY)) < 0)
26     {
27       perror (argv[1]);
28       return -2;
29     }
30
31   // consultem la mida de l'arxiu
32   if (stat (argv[1], &sb) < 0)
33     {
34       perror (argv[1]);
35       return -3;
36     }
37
38   // mapegem l'arxiu
39   if ((buff = mmap (NULL, sb.st_size, PROT_READ,
40                     MAP_PRIVATE, fd, 0)) == MAP_FAILED)
41     {
42       perror ("mmap");
```



```
43        return -4;
44      }
45
46    // ara ja podem tancar l'arxiu sense perill
47    close (fd);
48
49    // comptem quants d gits hi ha
50    for (counter = 0, digits = 0; counter < sb.st_size; counter++)
51      {
52        if (isdigit (buff[counter]))
53          digits++;
54      }
55    printf ("L'arxiu %s cont  %d car cters que s n d gits.\n",
56            argv[1], digits);
57
58    // eliminem el mapeig
59    munmap (buff, sb.st_size);
60    return EXIT_SUCCESS;
61  }
```

En aquest exemple es pot observar que un cop s'ha mapejat un arxiu a memòria, es pot tancar el seu file descriptor sense cap problema.

10.6.3 Mapejos d'arxius

Quins avantatges (i inconvenients) hi ha en treballar amb arxius servint-se de mapejos a memòria?

1. Pot resultar més ràpid: les crides a `read` i `write` (i similars) acostumen a necessitar dues transferències de dades: una de l'arxiu a la *cache* (buffer) del sistema operatiu, i una altra d'aquesta *cache* al buffer que el procés subministra a `read` o `write`. Amb la funció `mmap` s'elimina aquesta segona transferència.

2. Estalvia memòria: quan s'empra `read` o `write` (i similars), es mantenen dos buffers: un a nivell de sistema operatiu, i un altre a nivell de procés (d'aplicació). Usant `mmap`, hi ha un sol buffer compartit entre el nucli de l'operatiu i l'aplicació. I si a més a més el mapeig és compartit entre diversos processos, llavors poden compartir el mateix buffer, estalviant encara més memòria.

3. Per a arxius petits o per a poques operacions sobre l'arxiu, un mapeig de memòria pot ser més ineficient, ja que involucra:

   a) Obrir l'arxiu: una crida al sistema, amb el sobrecost que suposa.
   b) Obtenir la seva mida: una altra crida al sistema.
   c) Crear el mapeig: una altra crida al sistema.
   d) Provocar un *page fault* (ja que el mapeig no es fa realment efectiu fins la primera operació de lectura o escriptura), que comporta el seu sobrecost.

e) Desmapejar l'arxiu: una altra crida al sistema.

f) Actualització del hardware involucrat (el *memory management unit* i el *translation lookaside buffer*), amb el seu sobrecost.

Els avantatges d'accedir a fitxers amb mapeig de memòria es noten més si es fan accessos no seqüencials als arxius.

Hi ha una altra utilitat dels mapejos a memòria: fer-los servir com a mecanisme de comunicació entre processos. Un mapeig anònim pot servir per a aquest propòsit, però té l'inconvenient que només permet comunicar un procés amb el seu pare o els seus fills. En canvi, un mapeig compartit de fitxer permet comunicar processos que no tenen cap mena de relació. A més a més, pel fet de ser un mapeig sobre un fitxer, el contingut de la memòria compartida és persistent, és a dir, sobreviu a la finalització de les aplicacions i als reinicis del sistema. De totes maneres, cal tenir en compte que no es proveeix, automàticament, cap mecanisme de sincronització entre els processos que es comuniquen.

Queda pendent per comentar un aspecte. Ja s'ha dit que abans de mapejar un arxiu, cal haver-lo obert. Com es combinen les proteccions especificades al paràmetre `prot` amb la manera com s'ha obert l'arxiu (`O_RDONLY` i similars)?

- Si l'arxiu s'ha obert amb `O_RDWR`, llavors és vàlid qualsevol valor de `prot`.
- Si l'arxiu s'ha obert amb `O_WRONLY`, llavors qualsevol valor de `prot` és invàlid. El motiu és que moltes arquitectures de hardware no permeten que a una zona de memòria s'hi pugui escriure però no llegir, la qual cosa és incompatible amb `O_WRONLY`.
- Si l'arxiu s'ha obert amb `O_RDONLY`, llavors cal tenir en compte si es tracta d'un mapeig privat o compartit. Per a un mapeig privat, és vàlid qualsevol valor de `prot`, ja que els canvis que es puguin fer a la regió de memòria mapejada no es transfereixen a l'arxiu. Si el mapeig és compartit, llavors només són vàlids `PROT_READ`, `PROT_EXEC` o els dos a la vegada (`PROT_READ|PROT_EXEC`).

10.6.4 Mapejos anònims

Els mapejos anònims privats tenen una finalitat molt similar a la funció `malloc`: obtenir memòria. De fet, la llibreria de C de Linux fa que `malloc` cridi `mmap` per a peticions de més de 128 KB de memòria (tot i que aquest valor és configurable). Cal recordar que l'argument `fd` de `mmap` ha de valdre -1.

Per exemple, per demanar 1 MB de memòria:

```
char *addr = mmap (NULL, 1048576, PROT_READ | PROT_WRITE,
                   MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

Si el mapeig és compartit, llavors la zona de memòria queda compartida entre el pare i els fills que es creïn a continuació i abans d'eliminar el mapeig:


```
char *addr = mmap (NULL, 1048576, PROT_READ | PROT_WRITE,
                   MAP_SHARED | MAP_ANONYMOUS, -1, 0);
```

10.6.5 Sincronització de mapejos a memòria

El sistema operatiu garanteix que, per a un mapeig d'arxiu compartit, els canvis fets a la memòria quedaran reflectits a l'arxiu, però no indica quan es farà aquesta actualització. La funció `msync` permet especificar quan es faran:

```
#include <sys/mman.h>

int msync (void *addr, size_t length, int flags);
```

El paràmetre `addr` és l'adreça d'un mapeig a memòria, i `length` és la quantitat de bytes afectats per la sincronització. Hi ha dos possibles valors per a `flags`:

`MS_SYNC` La funció queda bloquejada fins que tots els canvis fets a la regió de memòria s'han reflectit a disc.

`MS_ASYNC` Els canvis fets a la regió de memòria es reflectiran a disc en algun altre moment, però seran visibles a d'altres processos que llegeixin de l'arxiu.

Dit d'una altra manera: `MS_SYNC` fa que la regió de memòria estigui sincronitzada amb l'arxiu a disc, mentre que `MS_ASYNC` fa que la regió de memòria estigui sincronitzada amb la *cache* del sistema operatiu.

10.7 La funció *fcntl*

La funció `fcntl` està pensada per a operacions que no estan incloses en cap dels altres mecanismes existents; per dir-ho d'alguna manera, `fcntl` actua com a "calaix de sastre" — juntament amb la funció `ioctl`, tot i que aquesta última té un àmbit d'actuació més ampli que `fcntl`.

La funció `fcntl` està declarada de la següent manera:

```
#include <fcntl.h>

int fcntl (int fd, int cmd, ...);
```

on `fd` és el file descriptor sobre el qual tindrà lloc l'operació especificada per `cmd`. En aquest apartat es descriuran només dues operacions; a mesura que s'avanci en el material en d'altres capítols s'aniran presentant més operacions.

La primera opció consisteix en obtenir els *flags* relatius a un arxiu obert (els mateixos que s'especifiquen a la funció `open`), i la segona consisteix en modificar-los. Es veuran ambdues operacions amb un exemple:

```
int fd, flags;
flags = fcntl (fd, F_GETFL);
if (flags == -1)
  perror ("fcntl");
else
  {
     flags |= O_APPEND;
     if (fcntl (fd, F_SETFL, flags) == -1)
       perror ("fcntl");
  }
```

10.8 La funció *epoll*

La funció `epoll` és una funció exclusiva de Linux, introduïda a la versió 2.6 del kernel. Engloba les funcionalitats de `select` i `poll` en una sola funció, a més de presentar els següents avantatges:

- El seu rendiment escala molt millor en presència de gran quantitat de files descriptors (centenars i milers).
- Permet notificar activitat en un file descriptor quan arriben *noves* dades, no quan *hi ha* dades.
- Quan hi ha activitat en un file descriptor, `epoll` pot retornar el file descriptor en qüestió o bé un punter, un enter de 32 bits o un enter de 64 bits, la qual cosa pot ser convenient en segons quins casos.

10.8.1 Inicialització

El primer pas és crear un file descriptor amb una de les dues funcions següents:

```
#include <sys/epoll.h>

int epoll_create (int size);
int epoll_create1 (int flags);
```

Les dues funcions retornen un file descriptor, que serà el que s'usarà per a la resta de funcions relacionades amb `epoll`. L'única diferència entre les dues funcions (a part que `epoll_create1` només està disponible a partir del kernel 2.6.27) és en el significat de l'argument que reben. En el cas de la funció `epoll_create` es tracta d'una estimació de la quantitat de file descriptors que es voldran monitoritzar, tot i que actualment el kernel no usa aquesta informació. En el segon cas, l'argument val zero o bé `EPOLL_CLOEXEC`, que fa el que descriptor retornat es tanqui, automàticament, quan el procés crida alguna de les funcions de la família `exec`.

10.8.2 Afegir file descriptors

Un cop s'ha cridat una de les dues funcions, es pot procedir a afegir file descriptors a la llista d'interès (que guarda el kernel). A diferència de `select` o `poll`, en què els file descriptors a monitoritzar s'indiquen (o afegeixen) cada cop que es crida `select` o `poll`, en el cas d'`epoll` només s'afegeixen un sol cop. Per això `epoll` rendeix molt millor quan hi ha molts file descriptors a monitoritzar que no `select` o `poll`.

Per a afegir file descriptors s'usa la funció `epoll_ctl`:

```
#include <sys/epoll.h>

int epoll_ctl (int efd, int op, int fd, struct epoll_event *ev);
```

El significat de cada argument és el següent:

`efd` File descriptor retornat per `epoll_create` o `epoll_create1`.

`op` Operació que es vol realitzar. Per tal d'afegir un file descriptor, `EPOLL_CTL_ADD`.

`fd` File descriptor que es vol afegir a la llista d'interès.

`event` Punter a un objecte que indicarà de quina manera es vol monitoritzar el file descriptor indicat per `fd`.

El tipus `struct epoll_event` està definit de la següent manera:

```
typedef union epoll_data {
  void        *ptr;
  int          fd;
  uint32_t     u32;
  uint64_t     u64;
} epoll_data_t;

struct epoll_event {
  uint32_t     events;      /* Epoll events */
  epoll_data_t data;        /* User data variable */
};
```

El camp `events` indica quins events es volen monitoritzar (per exemple, només lectura, només escriptura, lectura i escriptura, etc). El camp `data` serveix per indicar quin valor cal retornar quan hi hagi activitat en aquell file descriptor. Pot ser un punter, un file descriptor, un enter de 32 bits o un de 64 bits. Al ser una unió, només es pot fer servir *un i només un* dels camps de `epoll_data_t`.

10.8.3 Events que es poden monitoritzar

Els principals són els següents (la pàgina manual de `epoll_ctl` mostra totes les opcions existents):

`EPOLLIN` Monitoritzar el file descriptor per lectura. Es considerarà que hi ha activitat en aquest file descriptor quan hi hagi dades per a llegir.

`EPOLLOUT` Monitoritzar el file descriptor per escriptura. Es considerarà que hi ha activitat en aquest file descriptor quan es puguin escriure dades al file descriptor sense que el procés quedi bloquejat.

`EPOLLET` Monitoritzar el file descriptor per "nova activitat". Vegeu l'apartat 10.8.6.

`EPOLLERR` Indica que hi ha hagut algun error en l'activitat d'aquest file descriptor.

`EPOLLHUP` Indica que una connexió s'ha tancat.

Quan s'afegeixen file descriptors no cal especificar `EPOLLHUP` ni `EPOLLERR`, ja que el sistmea operatiu sempre monitoritza aquests dos events.

10.8.4 Esperar events

Un cop hem seleccionat quins files descriptors volem que *epoll* monitoritzi i quina activitat volem monitoritzar, per tal de ser informats dels events disposem de la funció *epoll_wait*, declarada de la següent manera:

```
#include <sys/epoll.h>

int epoll_wait(int epfd, struct epoll_event *events,
               int maxevents, int timeout);
```

on el significat de cada paràmetre és el següent:

`epfd` És el file descriptor retornat per *epoll_create* (o similar) i usat per a afegir file descriptors a la llista d'interès d'epoll.

`events` Array (o punter a una zona de memòria) que contindrà espai perquè *epoll_wait* indiqui quins file descriptors tenen activitat i de quina activitat es tracta. Aquest array o zona de memòria té, almenys, *maxevents* elements.

`maxevents` Nombre mínim d'elements que conté l'array o zona de memòria apuntada per *events*. La funció *epoll_events* no indicarà més de *maxevents* events.

`timeout` Indica, en microsegons, el timeout màxim que esperarà *epoll_wait*. Si és zero, no hi haurà espera en absolut; si és -1, l'espera serà indefinida (és a dir, sense timeout).

La funció *epoll_wait* retorna el número d'events que hi ha hagut, zero si no n'hi ha hagut cap, i -1 en cas d'error.

10.8.5 Exemple

L'exemple que es presenta és una petita adaptació del de la pàgina manual d'`epoll` (secció 7):

```
/*
 * In this example, listener is a nonblocking socket on
 * which listen(2) has been called. The function do_use_fd()
 * uses the new ready file descriptor until EAGAIN is
 * returned by either read(2) or write(2). An event-driven
 * state machine application should, after having received
 * EAGAIN, record its current state so that at the next
 * call to do_use_fd() it will continue to read(2) or
 * write(2) from where it stopped before.
 */

#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/epoll.h>
#include <sys/socket.h>

#define MAX_EVENTS 10

/* WARNING these functions ARE NOT implemented */
extern int setnonblocking (int);
extern int do_use_fd (int);

int
main (void)
{
  struct epoll_event ev, events[MAX_EVENTS];
  int listen_sock, conn_sock, nfds, epollfd, n;
  struct sockaddr_in local;
  socklen_t addrlen = sizeof (struct sockaddr_in);

  /* Set up listening socket, 'listen_sock' (socket(),
     bind(), listen()) */

  epollfd = epoll_create (10);
  if (epollfd == -1)
    {
      perror ("epoll_create");
      exit (EXIT_FAILURE);
    }

  ev.events = EPOLLIN;
  ev.data.fd = listen_sock;
  if (epoll_ctl (epollfd, EPOLL_CTL_ADD, listen_sock, &ev) == -1)
    {
```

```
46        perror ("epoll_ctl: listen_sock");
47        exit (EXIT_FAILURE);
48      }
49
50    while (1)
51      {
52        nfds = epoll_wait (epollfd, events, MAX_EVENTS, -1);
53        if (nfds == -1)
54          {
55            perror ("epoll_pwait");
56            exit (EXIT_FAILURE);
57          }
58
59        for (n = 0; n < nfds; ++n)
60          {
61            if (events[n].data.fd == listen_sock)
62              {
63                conn_sock = accept (listen_sock,
64                                    (struct sockaddr *) &local, &addrlen);
65                if (conn_sock == -1)
66                  {
67                    perror ("accept");
68                    exit (EXIT_FAILURE);
69                  }
70                setnonblocking (conn_sock);
71                ev.events = EPOLLIN | EPOLLET;
72                ev.data.fd = conn_sock;
73                if (epoll_ctl (epollfd, EPOLL_CTL_ADD, conn_sock, &ev) == -1)
74                  {
75                    perror ("epoll_ctl: conn_sock");
76                    exit (EXIT_FAILURE);
77                  }
78              }
79            else
80              do_use_fd (events[n].data.fd);
81          }
82      }
83  }
```

10.8.6 Notificació per nova activitat

L'exemple següent és una versió retallada de la solució d'una de les sessions del curs 2012–2013. Es tractava de dissenyar un servidor que acceptés connexions de diferents clients. Els clients enviaven trames de mida fixa — concretament d'onze bytes. Per simplicitat, s'ha eliminat la implementació d'alguna de les funcions, ja que no tenien res a veure amb el contingut d'aquest capítol.

És un exemple una mica llarg, però que ens donarà l'oportunitat de comentar com funciona la notificació per nova activitat i alguns aspectes relacionats amb la

lectura de sockets.

```
#ifndef _GNU_SOURCE
#define _GNU_SOURCE
#endif

#include <arpa/inet.h>
#include <assert.h>
#include <errno.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <stdarg.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/epoll.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <unistd.h>

typedef struct
{
  char *name;
  int quantity;
} entry_t;

typedef struct
{
  char *fname;
  uint16_t port;
} args_t;

static int server_parse_args (int argc, char *argv[], args_t * args);
extern int server_load_config (const char *fname, entry_t ** entries);
static int server_init (uint16_t port);
static int server_main_loop (int tcp_fd, entry_t * entries, int q_entries);
static int server_new_connection (int epoll_fd, int tcp_fd);
static int server_new_frame (int sock_fd, entry_t * entries, int q_entries);
extern int server_reply_frame (int sock_fd, entry_t * entries,
                               char buffer[11], int q_entries);

int
main (int argc, char *argv[])
{
  /* we pick the TCP port to open and the stock file */
  args_t args = { NULL, 0 };
  if (server_parse_args (argc, argv, &args) < 0)
```

```
49      return -1;
50
51    /* we load the stock file to memory */
52    entry_t *entries = NULL;
53    int q_entries = server_load_config (args.fname, &entries);
54    if (q_entries < 0)
55      return -1;
56    assert (entries);
57
58    /* we open the TCP port */
59    int tcp_fd = server_init (args.port);
60    if (tcp_fd < 0)
61      return -1;
62
63    return server_main_loop (tcp_fd, entries, q_entries);
64  }
65
66  static int
67  server_parse_args (int argc, char *argv[], args_t * args)
68  {
69    assert (argc > 0);
70    assert (argv);
71    assert (args);
72
73    if (argc < 3)
74      {
75        printf ("Not enough arguments (have %d, expected 3)\n", argc);
76        return -1;
77      }
78
79    if (access (argv[2], F_OK) < 0)
80      {
81        printf ("Could not access %s: %s (errno = %d)\n", argv[2],
82                strerror (errno), errno);
83        return -1;
84      }
85
86    int port = atoi (argv[1]);
87    if (port < 1 || port > 65535)
88      {
89        printf ("%s is not a valid TCP port\n", argv[1]);
90        return -1;
91      }
92
93    args->fname = strdup (argv[2]);
94    args->port = port;
95    return 0;
96  }
97
```



```
98  static int
99  server_init (uint16_t port)
100 {
101   assert (port > 0);
102
103   int fd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
104   if (fd < 0)
105     {
106       printf ("socket: %s (errno = %d)\n", strerror (errno), errno);
107       return -1;
108     }
109
110   struct sockaddr_in addr = { AF_INET, 0, {0}, {0} };
111   addr.sin_port = htons (port);
112
113   int opt = 1;
114   setsockopt (fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof (opt));
115
116   if (bind (fd, (void *) &addr, sizeof (addr)) < 0)
117     {
118       printf ("Could not bind to %s:%hu: %s (errno = %d, fd = %d)\n",
119               inet_ntoa (addr.sin_addr), ntohs (addr.sin_port),
120               strerror (errno), errno, fd);
121       close (fd);
122       return -1;
123     }
124
125   listen (fd, 5);
126   return fd;
127 }
128
129 static int
130 server_main_loop (int tcp_fd, entry_t * entries, int q_entries)
131 {
132   assert (tcp_fd > 0);
133   assert (entries);
134   assert (q_entries > 0);
135
136   /* epoll is better than select */
137   int epoll_fd = epoll_create1 (0);
138   if (epoll_fd < 0)
139     {
140       printf ("epoll_create1: %s (errno = %d)\n", strerror (errno), errno);
141       return -1;
142     }
143
144   struct epoll_event ev;
145   ev.data.fd = tcp_fd;
146   ev.events = EPOLLIN;
```

```
147
148   if (epoll_ctl (epoll_fd, EPOLL_CTL_ADD, tcp_fd, &ev) < 0)
149     {
150       printf ("EPOLL_CTL_ADD %d: %s (errno = %d, epoll_fd = %d)\n", tcp_fd,
151               strerror (errno), errno, epoll_fd);
152       close (epoll_fd);
153       return -1;
154     }
155
156   int n_ev = 1;
157   struct epoll_event *evs = calloc (n_ev, sizeof (*evs));
158
159   while (1)
160     {
161       int n = epoll_wait (epoll_fd, evs, n_ev, -1);
162       switch (n)
163         {
164         case -1:
165           if (errno == EINTR)
166             continue;
167           printf ("epoll_wait: %s (errno = %d, epoll_fd = %d)\n",
168                   strerror (errno), errno, epoll_fd);
169           return -1;
170
171         case 0:
172           printf ("epoll_wait returned zero (epoll_fd = %d)\n", epoll_fd);
173           return -1;
174         }
175
176       for (int i = 0; i < n; i++)
177         {
178           if (evs[i].data.fd == tcp_fd)
179             {
180               if (server_new_connection (epoll_fd, tcp_fd) > 0)
181                 {
182                   n_ev++;
183                   evs = realloc (evs, sizeof (*evs) * n_ev);
184                 }
185             }
186           else if (server_new_frame (evs[i].data.fd, entries, q_entries) < 0)
187             {
188               n_ev--;
189               evs = realloc (evs, sizeof (*evs) * n_ev);
190             }
191         }
192     }
193
194   /* not reached */
195   return -1;
```



```
196 }
197
198 static int
199 server_new_connection (int epoll_fd, int tcp_fd)
200 {
201   struct sockaddr_in addr = { AF_INET, 0, {0}, {0} };
202   socklen_t len = sizeof (addr);
203
204   int new_fd = accept (tcp_fd, (void *) &addr, &len);
205   if (new_fd < 0)
206     {
207       printf ("accept: %s (errno = %d, tcp_fd = %d)\n", strerror (errno),
208               errno, tcp_fd);
209       return -1;
210     }
211
212   printf ("New connection from %s:%hu\n", inet_ntoa (addr.sin_addr),
213           ntohs (addr.sin_port));
214
215   struct epoll_event ev;
216   ev.data.fd = new_fd;
217   ev.events = EPOLLIN | EPOLLET;
218
219   if (epoll_ctl (epoll_fd, EPOLL_CTL_ADD, new_fd, &ev) < 0)
220     {
221       printf ("EPOLL_CTL_ADD %d: %s (errno = %d, epoll_fd = %d)\n", new_fd,
222               strerror (errno), errno, epoll_fd);
223       close (new_fd);
224       return -1;
225     }
226
227   return new_fd;
228 }
229
230 static int
231 server_new_frame (int sock_fd, entry_t * entries, int q_entries)
232 {
233   assert (sock_fd > 0);
234   assert (entries);
235   assert (q_entries > 0);
236
237   /* is there an error? */
238   int errcode;
239   socklen_t len = sizeof (int);
240
241   getsockopt (sock_fd, SOL_SOCKET, SO_ERROR, &errcode, &len);
242   if (errcode != 0)
243     {
244       struct sockaddr_in addr;
```

```
245       socklen_t aux = sizeof (addr);
246       getpeername (sock_fd, (void *) &addr, &aux);
247       printf ("For connection from %s:%hu: %s (errno = %d, fd = %d)\n",
248               inet_ntoa (addr.sin_addr), ntohs (addr.sin_port),
249               strerror (errno), errno, sock_fd);
250       close (sock_fd);
251       return -1;
252     }
253
254   int q;
255   char buffer[11];
256   ioctl (sock_fd, FIONREAD, &q);
257   while (q >= 11)
258     {
259       memset (buffer, 0, 11);
260       read (sock_fd, buffer, 11);
261       if (server_reply_frame (sock_fd, entries, buffer, q_entries) < 0)
262         {
263           close (sock_fd);
264           return -1;
265         }
266       ioctl (sock_fd, FIONREAD, &q);
267     }
268
269   return 0;
270 }
```

Explicació L'esquema general de l'aplicació és el següent:

- L'aplicació rep, per línia de comandes, un número de port TCP i el nom d'un arxiu de configuració.
- La funció que carrega l'arxiu de configuració s'ha eliminat de l'exemple.
- La funció *server_init* s'encarrega d'inicialitzar el servidor. La seva feina es limita a obrir un socket passiu per tal de rebre noves connexions.
- La funció *server_main_loop* conté el codi del servidor en règim permanent, és a dir, un cop s'ha inicialitzat i està a punt per a funcionar.

Analitzem en detall la funció *server_main_loop*.

- Crida *epoll_create1* per a obtenir un file descriptor amb el qual realitzar operacions amb la família de funcions *epoll*. Ja hem vist que és el primer pas per a treballar amb *epoll*.
- Afegeix, a la llista d'interès, el socket passiu per a rebre connexions. L'activitat que interessa monitoritzar són les noves connexions, per això el flag especificat és simplement `EPOLLIN`.

- Es crida, dins d'un bucle infinit, *epoll_wait* per a saber quina activitat ha tingut lloc.

El punt interessant és saber què passa quan es rep una nova connexió. D'això se n'encarrega la funció *server_new_connection*. La clau de la qüestió és que, a l'afegir el file descriptor que retorna la funció *accept*, els flags especificats són la combinació de `EPOLLIN` i `EPOLLET`. Què vol dir això? Doncs molt senzill. Cada cop que vinguin dades per la nova connexió, *epoll_wait* ho notificarà. Però no tornarà a notificar activitat fins que vinguin *noves* dades, encara que les anteriors no s'hagin llegit. Si només s'hagués especificat `EPOLLIN`, quan arribessin dades per la nova connexió la funció *epoll_wait* sempre indicaria activitat al file descriptor, perquè trobaria dades pendents de ser llegides.

Quin és l'avantatge d'aquest mètode? Per a l'exemple en concret que presentem, no cal guardar els file descriptors de les connexions en variables o estructures de dades (n'hi ha prou que estiguin a la llista d'interès de l'*epoll*), amb la qual cosa l'aplicació guanya en simplicitat. A més a més, tampoc cal tenir buffers per guardar el que es va llegint de file descriptor. Això ho aconseguim en dos passos: primer, gràcies al fet que *epoll_wait* ens avisa *únicament* quan hi ha *noves* dades al socket; segon, gràcies a l'ús de la *ioctl* `FIONREAD` (a la funció *server_new_frame*), que ens permet consultar quantes dades hi ha pendents de llegir en un file descriptor.

Capítol 11

Ampliació de senyals

11.1 Introducció

El capítol 3 va servir d'introducció als senyals i va exposar les tècniques fonamentals per a treballar amb ells. L'objectiu del present capítol és ampliar el material, presentant versions més refinades d'algunes de les eines d'aquell capítol (com per exemple la funció `sigaction`) i també introduir noves tècniques per a la recepció de senyals, com ara la funció `signalfd`.

Encara que el lector estigui convençut que en té prou llegint el capítol 3, creiem sincerament que val molt la pena llegir el present capítol, ja que s'hi fan esments importants que poden ajudar el lector a evitar cometre segons quins errors, i després de veure les eines que aquí presentem potser decideix que val la pena fer-les servir a la seva pràctica.

11.2 Captura de senyals

La funció `signal` (apartat 3.5) és un mecanisme ofert per l'estàndar de C (és dels pocs mecanismes que s'expliquen en aquest llibre que formen part del llenguatge), i pot ser suficient en alguns casos. En d'altres, però, és necessari un control més detallat sobre com s'ha d'executar el signal handler, i a tal fi es disposa de la funció `sigaction`.

11.2.1 Funció *sigaction*

Està declarada de la següent manera:

```
#include <signal.h>

int sigaction (int signum,
               const struct sigaction *act,
               struct sigaction *oldact);
```

El tipus `struct sigaction` serveix per a establir, a més del *signal handler*, altres aspectes de control sobre com s'ha de comportar el sistema davant la recepció del senyal `signum`. Els camps més importants d'aquest tipus són:

- El camp `sa_handler` és similar al paràmetre `func` de la funció `signal`. Apunta a una funció (que rep un enter i retorna `void`) que actuarà com a *signal handler* del senyal `signum`.
- El camp `sa_sigaction` també apunta a una funció que actuarà com a *signal handler* del senyal `signum`; la diferència és que aquesta funció rep tres paràmetres en lloc d'un:
  - Un paràmetre de tipus `int`
  - Un paràmetre de tipus `siginfo_t *`
  - Un paràmetre de tipus `void *`

  A l'apartat 11.3 s'explicarà, amb detall, quin significat tenen aquests paràmetres.
- El camp `sa_flags` conté flags per a controlar diversos aspectes; entre d'altres, especifica quin dels dos tipus de *signal handler* s'ha de fer servir (el camp `sa_handler` i `sa_sigaction` se solapen, per tant, a l'especificar-ne un se sobreescriu l'altre).
- El camp `sa_mask` indica els senyals que s'han de bloquejar mentre s'executi el *signal handler*. Per defecte, durant l'execució d'un *signal handler* per al senyal S, aquest senyal queda bloquejat (i es desbloqueja quan s'acaba l'execució del *signal handler*. Amb aquest camp es poden especificar quins senyals extra s'han de bloquejar durant l'execució del *signal handler*.

Els arguments de la funció `sigaction` són:

- L'argument `signum` indica per a quin senyal s'està especificant el *signal handler*, a més de la resta de paràmetres de control.
- L'argument `act` conté els nous paràmetres de control per al senyal `signum`. Es pot passar un punter `NULL` si no es volen modificar tals paràmetres.
- L'argument `oldact` contindrà els antics paràmetres de control del senyal `signum`. Es pot passar un punter `NULL` si no es volen saber.

Queda pendent saber quins flags hi ha disponibles i quin és el seu funcionament:

SA_NOCLDSTOP Per defecte, el senyal SIGCHLD es rep quan un procés fill finalitza la seva execució o quan s'atura. Si aquest flag està activat quan s'instal·la un *signal handler* per a SIGCHLD (és a dir, quan signum val SIGCHLD), llavors quan el procés fill aturi (sense finalitzar) la seva execució no es rebrà SIGCHLD.

SA_NOCLDWAIT Quan un procés fill es mor, queda en un estat de *zombie* fins que el procés pare executa una crida wait o equivalent. Si quan s'instal·la un *signal handler* per a SIGCHLD s'activa aquest flag, s'evita la creació de processos *zombies*. Una altra manera d'evitar els processos *zombies* és fer que s'ignori el senyal SIGCHLD.

SA_NODEFER (També anomenat SA_NOMASK) Per defecte, quan s'executa un *signal handler* per a un determinat senyal, aquest queda bloquejat durant l'execució del *signal handler*. Si s'activa aquest flag, el senyal no es bloqueja.

SA_RESETHAND (També anomenat SA_ONESHOT) Instal·la el *signal handler* per a un sol ús; és a dir: a la primera vegada que es rep, s'executarà el *signal handler*, però a partir de llavors quedarà restaurada l'acció per defecte.

SA_RESTART Indica que la recepció d'un senyal no ha d'interrompre l'execució de les crides al sistema. Vegeu l'apartat 11.3.3 per a una discussió més profunda sobre la interacció entre senyals i crides al sistema.

SA_SIGINFO Aquest és el flag que determina quin dels dos tipus de *signal handler* s'utilitzarà. Si el flag no està activat, s'empra el handler de l'estil ANSI C (aquell en què el signal handler només rep un paràmetre). Si aquest flag està activat, s'empra el signal handler de l'estil POSIX (aquell en què el signal handler rep tres paràmetres).

Exemple Com a exemple es mostrarà una possible implementació de la funció signal emprant sigaction:

```
1  #include <signal.h>
2
3  typedef void (*sighandler_t) (int);
4
5  sighandler_t
6  Signal (int signo, sighandler_t func)
7  {
8    struct sigaction act, oact;
9
10   act.sa_handler = func;
11   sigemptyset (&act.sa_mask);
12   act.sa_flags = 0;
13   if (signo == SIGALRM)
```

```
14      act.sa_flags |= SA_RESTART;
15    if (sigaction (signo, &act, &oact) < 0)
16      return SIG_ERR;
17    return oact.sa_handler;
18  }
```

11.3 Signal handlers

11.3.1 Introducció

Un *signal handler* és una funció que es crida automàticament quan un determinat senyal s'entrega al procés. La crida al *signal handler* interromp l'execució del procés, i quan la funció retorna, el procés segueix la seva execució al punt on havia estat interromput. El *signal handler* no s'executa al mateix temps que la resta del procés (no hi ha cap mena de concurrència ni paral·lelisme). No es pot indicar que, al cridar-se automàticament un *signal handler*, se li ha de passar un determinat argument.

Hi ha dos prototipus per als *signal handlers*:

```
void handler (int signum);
void handler (int signum, siginfo_t *info, void *context);
```

En ambdós casos, `signum` indica el senyal rebut. Els paràmetre `info` conté informació addicional sobre el senyal rebut i com s'ha rebut, i el paràmetre `context` es pot emprar per a obtenir detalls del context del procés en el moment de rebre el senyal. Escollir un o altre tipus de *signal handler* és simplement una qüestió de quines necessitats tingui el programador.

11.3.2 Disseny de *signal handlers*

A l'apartat 3.6 vam fer una introducció a alguns principis de disseny de signal handlers. En aquest apartat volem aprofundir en la relació que hi pot haver entre un signal handler i una variable global, ja que molts signal handlers acaben tenint, com a única finalitat, modificar variables globals.

Sigui el següent exemple:

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <signal.h>
4  #include <string.h>
5
6  static int exit_flag = 0;
7
8  static void
9  hdl (int sig)
10 {
11   exit_flag = 1;
12 }
```



```
13
14  int
15  main (void)
16  {
17    struct sigaction act;
18
19    memset (&act, '\0', sizeof (act));
20    act.sa_handler = &hdl;
21    if (sigaction (SIGTERM, &act, NULL) < 0)
22      {
23        perror ("sigaction");
24        return 1;
25      }
26
27    while (!exit_flag)
28      ;
29
30    return 0;
31  }
```

Aparentment, aquest programa fa un bucle infinit mentre no es rep el senyal `SIGTERM`. Quan es rep aquest senyal, el *signal handler* modifica una variable global, que fa que el bucle infinit s'acabi i es finalitzi l'aplicació. Cal deixar ben clar que aquesta no és, ni de lluny, la millor manera de fer que un procés s'esperi a la recepció d'un senyal, però il·lustra perfectament el problema que es vol descriure.

Si s'observa més atentament el bucle infinit, es veurà que la variable que el controla (`exit_flag`) no canvia dins del bucle. Per tant, el compilador pot decidir traduïr aquest bucle de la següent manera: carregar la variable en un registre i no tornar-la a llegir de memòria. Per tant, el bucle serà efectivament infinit, *fins i tot* quan el *signal handler* modifiqui la variable (la modificarà a memòria, no al registre). El programa, doncs, pot ser que no faci el que aparentment ha de fer.

En quines circumstàncies pot passar, això? Bàsicament depèn de si el compilador optimitza o no el codi (per defecte, el compilador `gcc` no optimitza). Per tal de veure l'efecte d'una forma més contundent, s'ofereix un tros del codi generat (en assembler, evidentment), pel compilador `gcc` (versió 4.5.1, concretament). Per a la versió no optimitzada, el codi assembler rellevant és el següent:

```
1           call        sigaction
2           testl        %eax, %eax
3           jns         .L6
4           movl         $.LC0, %edi
5           call         perror
6           movl         $1, %eax
7           jmp         .L4
8   .L6:
9           nop
10  .L5:
```

```
11          movl        exit_flag(%rip), %eax
12          testl        %eax, %eax
13          je         .L5
14          movl        $0, %eax
15  .L4:
16          leave
17          .cfi_def_cfa 7, 8
18          ret
19          .cfi_endproc
```

Es crida la funció `sigaction` per a instal·lar el *signal handler*. A continuació es comprova el valor de retorn (instruccions `testl` i `jns`); si és negatiu, es crida la funció `perror`. En cas que el codi d'error no sigui negatiu, se salta a `.L6`. La instrucció `movl` carrega un valor de memòria a un registre, i la següent instrucció `je` fa que si tal valor és zero, es torni a saltar a `.L5`. Aquí el detall clau és que, *en cada iteració*, el valor de la variable `exit_flag` es carrega de memòria.

A continuació s'ofereix el codi assembler per a la versió optimitzada (és a dir, compilant amb `-O3`):

```
1           call        sigaction
2           testl        %eax, %eax
3           js         .L9
4           movl        exit_flag(%rip), %eax
5           testl        %eax, %eax
6           jne        .L6
7   .L7:
8           jmp        .L7
9           .p2align 4,,10
10          .p2align 3
11  .L6:
12          xorl        %eax, %eax
13  .L4:
14          addq        $168, %rsp
15          .cfi_remember_state
16          .cfi_def_cfa_offset 8
17          ret
18          .p2align 4,,10
19          .p2align 3
20  .L9:
21          .cfi_restore_state
22          movl        $.LC0, %edi
23          call        perror
24          movl        $1, %eax
25          jmp        .L4
26          .cfi_endproc
```

En aquest cas, si el valor de retorn de `sigaction` és negatiu, se salta a `.L9`, que conté la crida a la funció `perror`. A continuació es carrega, en un registre, el valor

de la variable `exit_flag`. Si aquest valor és diferent de zero, se salta a `.L6`, i se segueixen executant instruccions fins a finalitzar el programa. Si no se salta a `.L6`, s'entra directament a `.L7`, on la primera instrucció que es troba és, directament, un salt incondicional a `.L7`: és a dir, un bucle infinit. Les iteracions són simplement salts, sense comprovar cap altra condició: la variable `exit_flag` *no es torna a llegir mai més*.

Com es pot resoldre aquest problema? La primera solució és no activar les opcions d'optimització del compilador, però hi ha ocasions en què això no és possible (especialment si interessa millorar el rendiment del programa). La segona solució, correcta, és indicar al compilador que la variable `exit_flag` pot canviar fora del bucle:

```
static volatile int exit_flag = 0;
```

Aquí la clau és el qualificador `volatile`. A continuació s'ofereix el tros rellevant de codi assembler per a la nova versió del codi font, havent activat les optimitzacions:

```
1         call        sigaction
2         testl        %eax, %eax
3         js        .L8
4         .p2align 4,,10
5         .p2align 3
6   .L5:
7         movl        exit_flag(%rip), %eax
8         testl        %eax, %eax
9         je        .L5
10        xorl        %eax, %eax
11  .L4:
12        addq        $168, %rsp
13        .cfi_remember_state
14        .cfi_def_cfa_offset 8
15        ret
16  .L8:
17        .cfi_restore_state
18        movl        $.LC0, %edi
19        call        perror
20        movl        $1, %eax
21        jmp        .L4
22        .cfi_endproc
```

Un cop s'ha comprovat el valor de retorn de la funció `sigaction`, s'entra a `.L5`, i la primera instrucció que hi ha és justament carregar de memòria a un registre el valor de la variable `exit_flag`. Si el valor d'aquesta variable és zero, se salta un altre cop a `.L5`, i per tant es torna a llegir la variable de memòria.

11.3.3 Senyals i crides al sistema

Imagini's el cas d'un procés que està executant una crida al sistema com ara `read`, esperant entrada de teclat o d'Internet, i mentre s'està esperant arriba un senyal que provoca l'execució d'un *signal handler*. Quan aquest finalitza la seva execució, en quin punt s'ha de seguir executant el procés? Per defecte, la crida al sistema retorna -1 amb un error `EINTR`, tot i que no s'interrompen totes les crides al sistema (per a més detalls es pot consultar l'apartat 10.5 de Stevens and Rago [2008]). Però al programador potser li interessa que se segueixi executant la crida al sistema.

Hi ha dues maneres de resoldre aquest problema:

1. Quan la crida al sistema retorna, comprovar si ha retornat per culpa d'un senyal (en cas afirmatiu, la variable global `errno`, declarada a `errno.h`, valdrà `EINTR`). Si és així, es pot tornar a cridar la crida al sistema.

2. A l'instal·lar un *signal handler*, especificar que, en cas de rebre un senyal, no s'han d'interrompre les crides al sistema. A tal fi, s'instal·la el *signal handler* amb la funció `sigaction` i activem el flag `SA_RESTART`.

L'inconvenient del primer mètode és que requereix una mica més de codi, però té l'avantatge que funciona amb totes les crides al sistema que són interrompudes per senyals. El segon mètode, en canvi, tot i que no requereix codi extra, no funciona amb totes les crides que s'interrompen per senyals. Les pàgines 443-444 de Kerrisk [2010] detallen, per a Linux, quines crides al sistema són automàticament reiniciades si `SA_RESTART` està activat.

11.4 Funcions *sigpending* i *sigsuspend*

```
#include <signal.h>

int sigpending (sigset_t *set);
int sigsuspend (const sigset_t *set);
```

- La funció `sigpending` fa que `set` apunti al conjunt de senyals que estan esperant ser atesos. És a dir, aquells senyals que s'han enviat al procés però que encara no s'han entregat.

- La funció `sisguspend` fa dues coses:

    1. Estableix, com a únics senyals bloquejats pel procés, els que formen part del conjunt apuntat per `set`. Això es fa de forma temporal, fins que la funció retorna.
    2. Adorm el procés fins que el procés rep un senyal pel qual ha d'executar un signal handler (o bé el procés ha de finalitzar la seva execució).

La funció `sigsuspend` retorna un cop s'ha executat el signal handler instal·lat per al senyal rebut. Un cop retorna, el procés té bloquejats els senyals que tenia bloquejats *abans* de cridar aquesta funció.

Quina utilitat pot tenir la funció `sigsuspend`? Suposi's el següent escenari (apartat 22.9 de Kerrisk [2010]):

1. Es bloqueja temporalment un senyal per tal que el seu *signal handler* no interrompi l'execució d'una regió crítica del codi.
2. Un cop executada la regió crítica, es desbloqueja el senyal i se suspèn l'execució del procés fins que el senyal sigui entregat al procés.

El següent exemple mostra una primera implementació d'aquest escenari:

```
1  // signals/sigsuspend.c
2  #include <signal.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6
7  void
8  exemple (void (*handler) (int))
9  {
10   sigset_t prevMask, intMask;
11   struct sigaction sa;
12
13   sigemptyset (&intMask);
14   sigaddset (&intMask, SIGINT);
15
16   sigemptyset (&sa.sa_mask);
17   sa.sa_flags = 0;
18   sa.sa_handler = handler;
19
20   if (sigaction (SIGINT, &sa, NULL) < 0)
21     {
22       perror ("sigaction");
23       exit (EXIT_FAILURE);
24     }
25
26   /* Block SIGINT prior to executing critical section.
27    * (At this point we assume that SIGINT is not
28    * already blocked) */
29   if (sigprocmask (SIG_BLOCK, &intMask, &prevMask) < 0)
30     {
31       perror ("sigprocmask SIG_BLOCK");
32       exit (EXIT_FAILURE);
33     }
34
```

```
35    /* Critical section: do some work here that must not be
36     * interrupted by the SIGINT handler */
37
38    /* End of critical section - restore old mask to unblock
39     * SIGINT */
40    if (sigprocmask (SIG_SETMASK, &prevMask, NULL) < 0)
41      {
42        perror ("sigprocmask SIG_SETMASK");
43        exit (EXIT_FAILURE);
44      }
45
46    /* BUG: what if SIGINT arrives now... */
47    pause ();
48  }
```

Com es pot observar, hi ha una finestra de temps entre que es desbloqueja el senyal i se suspèn l'execució del procés; en aquesta finestra de temps es podria cridar el *signal handler* i la funció `pause` bloquejaria el procés definitivament. La funció `sigsuspend` resol el problema, ja que *atòmicament* desbloqueja el senyal i suspèn l'execució del procés fins que el *signal handler* s'ha acabat d'executar.

11.5 Funcions *sigwait* i *sigwaitinfo*

A l'apartat anterior s'ha vist que la funció `sigsuspend` s'espera que s'hagi executat el *signal handler* corresponent al senyal que s'ha entregat, la qual cosa requereix, com és lògic, que el programador escrigui un *signal handler*. Hi ha una alternativa: emprar les funcions `sigwait` o `sigwaitinfo`, que eliminen la necessitat d'escriure un *signal handler*. Ara bé, convé haver bloquejat, prèviament, els senyals que s'esperen.

```
#include <signal.h>

int sigwait (const sigset_t *set, int *sig);
int sigwaitinfo (const sigset_t *set, siginfo_t *info);
```

Les dues funcions tenen la mateixa funcionalitat: adormen el procés (o *thread*) fins que arriba algun dels senyals presents al conjunt apuntat per `set`. L'única diferència és que:

- La funció `sigwait` indica el número de senyal que ha arribat amb el punter `sig`. Retorna zero si no hi ha hagut cap error. Si hi ha hagut un error, el valor de la funció és directament el codi de l'error (a diferència d'altres funcions que retornen -1 en cas d'error i a la variable global `errno` hi posen el codi d'error).

- La funció `sigwaitinfo` indica, amb el punter `info`, el número de senyal que ha arribat i informació addicional (l'apartat 11.2.1 dóna informació addicional sobre el tipus `siginfo_t`). Es pot passar un punter `NULL` a l'argument `info`. La funció retorna el número de senyal arribat. L'apartat 22.10 de Kerrisk [2010] dóna alguns exemples de com emprar aquesta funció.

Si el que es vol és imposar un cert *timeout* en l'espera, es pot emprar la funció següent:

```
#include <signal.h>

int sigtimedwait (const sigset_t *set,
                  siginfo_t *info,
                  const struct timespec *timeout);
```

on `struct timespec` està definit de la següent manera:

```
struct timespec
{
  long int tv_sec;              // segons
  long int tv_nsec;             // nanosegons
};
```

11.6 Funció *signalfd*

Fins al moment s'han vist dos grans enfocaments per a rebre senyals: instal·lar *signal handlers*, o bé usar funcions que esperen l'arribada de senyals. La funció `signalfd` ofereix un tercer mecanisme: rebre senyals "a través" de file descriptors. Aquesta funció està declarada de la següent manera:

```
#include <sys/signalfd.h>

int signalfd (int fd, const sigset_t *mask, int flags);
```

- El paràmetre `id` és un file descriptor que s'ha obert, anteriorment, mitjançant una crida a `signalfd`. Si el que es vol és obtenir un nou file descriptor, l'argument ha de valdre -1.
- El paràmetre `mask` apunta a un conjunt de senyals, que conté els senyals que es volen rebre a través del file descriptor retornat per `signalfd`.
- El paràmetre `flags` permet especificar-ne dues:

  `SFD_NONBLOCK` Especifica que el file descriptor no serà bloquejant.

  `SFD_CLOEXEC` Especifica que el file descriptor s'ha de tancar si el procés executa la funció `execve` o similars.

Un cop el file descriptor ha estat obert, quines operacions es poden fer amb ell?

- La funció `read` es pot fer servir per a llegir del file descriptor. El buffer de memòria és un array d'almenys un element d'un tipus que es descriurà més endavant. La funció `read` retornarà si s'ha rebut un signal dels especificats al conjunt apuntat per `mask`. Si no hi ha cap d'aquests signals pendents, llavors el procés s'adormirà (o bé la funció `read` retornarà `EAGAIN` si el file descriptor no és bloquejant).

- La funció `select` (i d'altres funcions com ara `poll`) consideren que el file descriptor està a punt per lectura quan el procés ha rebut un signal dels especificats al conjunt apuntat per `mask`.

- Per a tancar el file descriptor s'empra la funció `close`.

Tal com s'ha dit, la funció `read` permet *llegir* senyals a través del file descriptor. Cada cop que es crida la funció `read`, se li passa un punter a un element del tipus següent:

```
struct signalfd_siginfo
{
  uint32_t ssi_signo;   // signal number
  int32_t ssi_errno;    // error number (unused)
  int32_t ssi_code;     // signal code
  uint32_t ssi_pid;     // PID of the sender
  uint32_t ssi_uid;     // real UID of the sender
  int32_t ssi_fd;       // file descriptor (SIGIO)
  uint32_t ssi_tid;     // kernel timer ID (POSIX timers)
  uint32_t ssi_band;    // band event (SIGIO)
  uint32_t ssi_overrun; // POSIX timer overrun count
  uint32_t ssi_trapno;  // trap number that caused the signal
  int32_t ssi_status;   // exit status or signal (SIGCHLD)
  int32_t ssi_int;      // integer sent by sigqueue
  uint64_t ssi_ptr;     // pointer sent by sigqueue
  uint64_t ssi_utime;   // user CPU time consumed (SIGCHLD)
  uint64_t ssi_stime;   // system CPU time consumed (SIGCHLD)
  uint64_t ssi_addr;    // address that generated signal
  // (for hardware-generated signals)
  uint8_t ssi_pad[];    // pad size to 128 bytes
}
```

Cadascun d'aquests camps és similar al seu corresponent camp de `siginfo_t` (vegi's l'apartat 11.2.1).

Exemple Com a exemple, es presenta un programa que accepta els senyals `SIGINT` i `SIGQUIT`. El programa finalitza quan rep aquest últim senyal.


```
#include <sys/signalfd.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

#define handle_error(msg) \
           do { perror(msg); exit(EXIT_FAILURE); } while (0)

int
main (void)
{
  sigset_t mask;
  int sfd;
  struct signalfd_siginfo fdsi;
  ssize_t s;

  sigemptyset (&mask);
  sigaddset (&mask, SIGINT);
  sigaddset (&mask, SIGQUIT);

  /* Block signals so that they aren't handled
     according to their default dispositions */

  if (sigprocmask (SIG_BLOCK, &mask, NULL) == -1)
    handle_error ("sigprocmask");

  sfd = signalfd (-1, &mask, 0);
  if (sfd == -1)
    handle_error ("signalfd");

  for (;;)
    {
      s = read (sfd, &fdsi, sizeof (struct signalfd_siginfo));
      if (s != sizeof (struct signalfd_siginfo))
        handle_error ("read");

      if (fdsi.ssi_signo == SIGINT)
        {
          printf ("Got SIGINT\n");
        }
      else if (fdsi.ssi_signo == SIGQUIT)
        {
          printf ("Got SIGQUIT\n");
          exit (EXIT_SUCCESS);
        }
      else
        {
          printf ("Read unexpected signal\n");
```

```
50          }
51      }
52
53    return EXIT_SUCCESS;
54  }
```


Capítol 12

Sockets: eines avançades

12.1 Introducció

El capítol 7 va servir d'introducció al món dels sockets; el present capítol pretén completar i ampliar la informació presentada en aquell capítol. Es comença presentant un parell de funcions que poden ser molt útils de cara a connexions TCP, i a continuació es veuran algunes opcions de sockets, que permeten establir un control més refinat sobre la comunicació que s'estableix a través dels sockets. S'ha considerat interessant també presentar el protocol UDP i com funcionen els sockets en mode no bloquejant, i algunes operacions `ioctl` relacionades amb sockets (però les que tenen a veure amb les interfícies de xarxa es postposen per al capítol 13).

12.2 Funcions *getsockname* i *getpeername*

Per als protocols orientats a connexió (és a dir, TCP), a vegades interessa saber quina és l'adreça i port de cadascun dels extrems de la connexió. Tot i que la funció `connect` especifica l'adreça del destinatari, i la funció `bind` especifica la de l'origen, a vegades es deixa que sigui el sistema operatiu qui triï algun dels paràmetres, i no sempre es manté disponible la informació (per exemple, una variable global per a cada socket que guardi les adreces origen i destí). Les funcions `getsockname` i `getpeername` permeten obtenir aquesta informació.

12.2.1 Funció *getsockname*

Està declarada de la següent manera:

```
#include <sys/socket.h>

int getsockname (int sockfd, struct sockaddr *addr,
                 socklen_t *addrlen);
```

El primer paràmetre fa referència al socket (que pot ser tant actiu com passiu, és a dir, per a un client o per a un servidor). El segon paràmetre apunta a l'objecte que contindrà l'adreça del socket, tal com s'hauria passat a `bind`. I el tercer paràmetre és un punter a una variable, que s'haurà inicialitzat amb la mida de l'objecte apuntat pel segon paràmetre.

12.2.2 Funció *getpeername*

Està declarada de la següent manera:

```
#include <sys/socket.h>

int getpeername (int sockfd, struct sockaddr *addr,
                 socklen_t *addrlen);
```

El primer paràmetre fa referència al socket (que pot ser tant actiu com passiu, és a dir, per a un client o per a un servidor). El segon paràmetre apunta a l'objecte que contindrà l'adreça del socket, tal com s'hauria passat a `bind`. I el tercer paràmetre és un punter a una variable, que s'haurà inicialitzat amb la mida de l'objecte apuntat pel segon paràmetre.

12.2.3 Exemple

Com a exemple, es presenta una funció que rep un socket TCP d'una connexió establerta, i mostra les adreces i ports origen i destí.

```
1  #include <arpa/inet.h>
2  #include <stdio.h>
3
4  void
5  print_tcp_addresses (int sock_fd)
6  {
7    struct sockaddr_in src, dst;
8    socklen_t len = sizeof (struct sockaddr_in);
9
10   if (getsockname (sock_fd, (void *) &src, &len) < 0)
11     perror ("getsockname");
12
13   if (getpeername (sock_fd, (void *) &dst, &len) < 0)
14     perror ("getpeername");
15
16   printf ("%s:%hu --> ", inet_ntoa (src.sin_addr), ntohs (src.sin_port));
```

```
17    printf ("%s:%hu\n", inet_ntoa (dst.sin_addr), ntohs (dst.sin_port));
18  }
```

Es fan dues crides a `printf` perquè la funció `inet_ntoa` no és reentrant, ja que utilitza un array estàtic a on s'hi guarda la representació en ASCII de l'adreça IP. Si no es fes així, apareixeria la mateixa adreça IP com a origen i destí.

12.3 Opcions de sockets

Les opcions dels sockets són una manera per a refinar el control sobre la comunicació entre client i servidor. Hi ha una gran quantitat d'opcions, les més freqüents de les quals es comenten en aquest apartat. Per a consultar i establir opcions de socket hi ha les dues funcions següents:

```
#include <sys/socket.h>

int getsockopt (int sockfd, int level,
                int optname, void *optval,
                socklen_t *optlen);
int setsockopt (int sockfd, int level,
                int optname, const void *optval,
                socklen_t optlen);
```

`sockfd` És el socket pel qual es vol consultar o establir l'opció.

`level` Indica el tipus d'opció. Sol valdre `SOL_SOCKET` per a les opcions el nom de les quals comencen per `SO_`, i `IPPROTO_TCP` per a les opcions el nom de les quals comencen per `TCP_`.

`optname` És el nom de l'opció pròpiament dita, com ara `TCP_CORK`.

`optval` És un punter al nou valor que prendrà l'opció. En la majoria de vegades serà un enter, en d'altres serà alguna estructura.

`optlen` Per a `getsockopt`, punter a una variable que contindrà la mida de l'objecte apuntat per `optval`. Per a `setsockopt`, la mida de l'objecte apuntat per `optval`.

Tal com s'ha dit, aquí no es presenten totes les opcions disponibles, només les que s'han considerat més útils o interessants. El capítol 7 de Stevens et al. [2004] està dedicat exclusivament a opcions de sockets; també es poden consultar les pàgines manuals dels protocols per tal de conèixer els detalls específics d'un sistema operatiu (UNIX, evidentment) en concret.

12.3.1 Opció SO_ERROR

Aquesta opció serveix per saber si, per a un socket, hi ha algun error pendent (*pending error*). Tot i que les funcions que treballen amb sockets acostumen a indicar la presència d'errors (retornant -1 i modificant la variable global `errno`), hi ha ocasions en què cal (o pot ser molt útil) saber si hi ha algun error sense necessitat d'efectuar cap operació (per a un cas pràctic, vegeu l'apartat 12.5, en les connexions no bloquejants). L'opció de socket `SO_ERROR` ens permet fer-ho.

Exemple

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>

int
is_there_an_error (int sockfd)
{
  int codi;
  socklen_t len = sizeof (codi);

  if (getsockopt (sockfd, SOL_SOCKET, SO_ERROR, &codi, &len) < 0)
    perror ("getsockopt");

  printf ("Error: %s (codi %d)\n", strerror (codi), codi);
  return codi;
}
```

Nota L'opció `SO_ERROR` només es pot consultar, és a dir, no es pot cridar `setsockopt` amb l'opció `SO_ERROR`.

12.3.2 Opció SO_LINGER

Aquesta funció modifica el comportament de la funció `close` quan es tracta d'una connexió TCP. Per defecte, la funció `close` retorna immediatament; ara bé, si hi ha dades pendents de confirmar (és a dir, que s'han enviat però el receptor no ha confirmat la recepció), s'aniran reenviant fins que el receptor les confirmi, i llavors es tancarà la connexió. L'opció `SO_LINGER` modifica aquest comportament, servint-se de la següent estructura:

```
struct linger
{
  int l_onoff;
  int l_linger;
};
```


Per a activar l'opció cal que `l_onoff` valgui 1; en cas contrari, es desactiva l'opció. El camp `l_linger` actua de *timeout*:

- Si val zero, llavors el sistema no tanca la connexió amb els segments FIN, sinó que envia un segment RST, de manera que si hi havia dades perdents de confirmar, es poden perdre. A més a més, la connexió no passarà per l'estat TIME_WAIT — però vegeu [Stevens et al., 2004, pàgina 203].

- Si no val zero, llavors el valor s'interpreta com un *timeout*, en segons. Durant aquest temps, la funció `close` bloqueja el procés, esperant que les dades pendents arribin (i arribi també la confirmació de recepció, els segments ACK). Si ha passat aquest temps i no s'ha pogut confirmar tot, llavors s'envia el segment RST i es descarten les dades pendents. En aquest cas, la funció `close` retornarà -1 amb el codi d'error `EWOULDBLOCK`.

  Si el socket està en mode no bloquejant, `close` retorna immediatament, independentment del valor de `l_linger`.

Per als detalls complets del funcionament d'aquesta opció es recomana llegir atentament [Stevens et al., 2004, pàgines 202–207].

Exemple

```
1  #include <stdio.h>
2  #include <sys/socket.h>
3
4  void
5  activate_linger (int sock_fd, int timeout)
6  {
7    struct linger l = { 1, timeout };
8
9    if (setsockopt (sock_fd, SOL_SOCKET, SO_LINGER, &l, sizeof (l)) < 0)
10     perror ("setsockopt SO_LINGER");
11 }
```

12.3.3 Opcions SO_RCVBUF i SO_SNDBUF

Quan un procés escriu en un socket, el que fa és col·locar les dades en un buffer que el sistema operatiu guarda per a cada socket, la qual cosa no significa que s'enviï un paquet al destinatari de les dades — cada protocol té les seves "normes" pel que fa a l'enviament de dades (pel cas del protocol TCP, per exemple, hi ha mecanismes de control de congestió com ara *slow start*, *congestion avoidance*, *fast recovery* i *fast retransmit*). Igualment, quan arriben dades procedents de la xarxa, el sistema operatiu les guarda en un buffer, fins que el procés les llegeix (en TCP, per exemple, la mida d'aquest buffer s'empra per a determinar l'*advertised window* que s'anuncia durant l'establiment de la connexió).

La mida d'aquests buffers la decideix el sistema operatiu, i un procés la pot canviar (dins d'uns certas marges) amb les opcions `SO_RCVBUF` (buffer de recepció, per a les lectures) i `SO_SNDBUF` (buffer d'enviament, per a les escriptures). Per a un socket TCP, el canvi de mida s'ha de fer al moment oportú:

- Per al client, abans de cridar `connect`.
- Per al servidor, abans de cridar `listen`. Tots els sockets que retorni la funció `accept` heretaran les mides dels buffers.[1]

Linux sempre multiplica per dos el valor especificat per l'opció de socket. És a dir, si especifiquem una mida de 10 KB amb `setsockopt`, al cridar `getsockopt` ens retornarà una mida de 20 KB. Per a `SO_RCVBUF` la mida mínima (després de doblar) és de 256 bytes, i per a `SO_RCVBUF` és de 2048 bytes. Hi ha també màxims, que depenen de la configuració del sistema (vegeu la pàgina manual per a `socket`, secció 7: `man 7 socket`).

Exemple El següent exemple és un programa que rep dos arguments per línia de comandes: la mida en bytes del buffer d'escriptura, i la mida en bytes del buffer de lectura. El programa crea un socket TCP i modifica la mida dels buffers perquè corresponguin amb els valors passats per línia de comandes. A continuació consulta la mida dels buffers i les mostra per pantalla. Primer es mostra el codi font, i després un exemple d'execució.

```
1  #include <netinet/in.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <sys/socket.h>
5
6  int
7  main (int argc, char *argv[])
8  {
9    if (argc < 3)
10     {
11       fprintf (stderr, "Falten arguments: SO_SNDBUF SO_RCVBUF\n");
12       return -1;
13     }
14
15   int snd_size = atoi (argv[1]);
16   int rcv_size = atoi (argv[2]);
17
18   if (snd_size < 2048)
19     {
20       printf ("%s  s un valor inv lid\n", argv[1]);
21       return -1;
```

[1] Vegeu [Kerrisk, 2010, apartat 61.11].


```
22      }
23    if (rcv_size < 256)
24      {
25        printf ("%s  s un valor inv lid\n", argv[2]);
26        return -1;
27      }
28
29    int sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
30    if (sockfd < 0)
31      {
32        perror ("socket");
33        return -1;
34      }
35
36    if (setsockopt (sockfd, SOL_SOCKET, SO_SNDBUF, &snd_size, sizeof (int)) < 0)
37      {
38        perror ("setsockopt SO_SNDBUF");
39        return -1;
40      }
41
42    if (setsockopt (sockfd, SOL_SOCKET, SO_RCVBUF, &rcv_size, sizeof (int)) < 0)
43      {
44        perror ("setsockopt SO_RCVBUF");
45        return -1;
46      }
47
48    socklen_t len = sizeof (int);
49    if (getsockopt (sockfd, SOL_SOCKET, SO_SNDBUF, &snd_size, &len) < 0)
50      {
51        perror ("getsockopt SO_SNDBUF");
52        return -1;
53      }
54    else
55      printf ("Mida SO_SNDBUF: %d bytes\n", snd_size);
56
57    if (getsockopt (sockfd, SOL_SOCKET, SO_RCVBUF, &rcv_size, &len) < 0)
58      {
59        perror ("getsockopt SO_RCVBUF");
60        return -1;
61      }
62    else
63      printf ("Mida SO_RCVBUF: %d bytes\n", rcv_size);
64  }
```

```
$ ./a.out 8192 16384
Mida SO_SNDBUF: 16384 bytes
Mida SO_RCVBUF: 32768 bytes
$
```

12.3.4 Opcions SO_RCVLOWAT i SO_SNDLOWAT

Aquestes opcions permeten alterar la "sensibilitat" d'algunes funcions (les que multiplexen file descriptors, com ara `select` i `poll`) quan operen sobre sockets.

- L'opció `SO_RCVLOWAT` especifica quants bytes hi ha d'haver al buffer d'un socket per tal que es consideri que aquell file descriptor està a punt per llegir.
- L'opció `SO_SNDLOWAT` especifica quants bytes lliures hi ha d'haver al buffer d'un socket per tal que es consideri que aquell file descriptor està a punt per a escriure.

Això és el que diu la teoria, però no sempre es correspon amb la pràctica. En Linux, la pàgina manual de `socket` (secció 7) indica que les funcions de multiplexació (com ara `select` i `poll`) ignoren els valors d'aquestes dues opcions, però en canvi funcions com `read` sí en fan cas:

> Specify the minimum number of bytes in the buffer until the socket layer will pass the data to the protocol (`SO_SNDLOWAT`) or the user on receiving (`SO_RCVLOWAT`). These two values are initialized to 1. `SO_SNDLOWAT` is not changeable on Linux (`setsockopt(2)` fails with the error `ENOPROTOOPT`). `SO_RCVLOWAT` is changeable only since Linux 2.4. The `select(2)` and `poll(2)` system calls currently do not respect the `SO_RCVLOWAT` setting on Linux, and mark a socket readable when even a single byte of data is available. A subsequent read from the socket will block until `SO_RCVLOWAT` bytes are available.

Tot això contrasta amb el que s'explica a [Stevens et al., 2004, pàgines 209–210] i amb d'altres implementacions de UNIX, com ara FreeBSD, que en la seva pàgina manual (`setsockop(2)`) indica que:

> `SO_SNDLOWAT` is an option to set the minimum count for output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed. A `select(2)` operation testing the ability to write to a socket will return true only if the low water mark amount could be processed. The default value for `SO_SNDLOWAT` is set to a convenient size for network efficiency, often 1024. `SO_RCVLOWAT` is an option to set the minimum count for input operations. In general, receive calls will block until any (non-zero) amount of data is received, then return with the smaller of the amount available or the amount requested. The default value for `SO_RCVLOWAT`

> is 1. If SO_RCVLOWAT is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different from that which was returned.

12.3.5 Opcions SO_RCVTIMEO i SO_SNDTIMEO

Aquestes dues opcions permeten especificar un *timeout* en l'enviament i la recepció de dades:

- L'opció SO_RCVTIMEO afecta les següents funcions: `read`, `readv`, `recv`, `recvfrom` i `recvmsg`.
- L'opció SO_SNDTIMEO afecta les següents funcions: `write`, `writev send`, `sendto` i `sendmsg`.

Per a qualsevol d'aquestes funcions, el procés queda bloquejat fins que arriben dades (cas d'una lectura) o hi ha espai al buffer per a escriure les dades (cas d'una escriptura). Si quan expira el timeout no han arribat dades (o segueix sense haver-hi prou espai al buffer), llavors la funció corresponent retorna -1 i el codi d'error és EAGAIN, com si es tractés d'un socket no bloquejant (per a sockets no bloquejants vegeu l'apartat 12.5). Si el temps de timeout és zero, llavors el que es fa és desactivar l'opció.

Les funcions `select` (i família, com ara `pselect`, `poll`, `epoll_wait` i similars no es veuen afectades, com tampoc `connect` i `accept`.

Exemple El següent exemple consisteix en un programa que rep dos paràmetres per línia de comandes: una adreça IP i un port TCP, al qual l'aplicació s'intenta connectar. S'envia una frase al destinatari, del qual s'espera resposta durant 5 segons.

```
1  #include <arpa/inet.h>
2  #include <netinet/in.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/socket.h>
6  #include <time.h>
7  #include <unistd.h>
8
9  int
10 main (int argc, char *argv[])
11 {
12   if (argc < 3)
13     {
14       fprintf (stderr, "Falta especificar IP i port TCP\n");
15       return -1;
```

```
16      }
17
18    int sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
19    if (sockfd < 0)
20      {
21        perror ("socket");
22        return -1;
23      }
24
25    struct sockaddr_in addr = { AF_INET, 0, {0}, {0} };
26    addr.sin_port = htons (atoi (argv[2]));
27    if (inet_aton (argv[1], &addr.sin_addr) == 0)
28      {
29        fprintf (stderr, "%s no  s una adre a IP v lida\n", argv[1]);
30        return -1;
31      }
32
33    if (connect (sockfd, (void *) &addr, sizeof (addr)) < 0)
34      {
35        perror ("connect");
36        return -1;
37      }
38
39    /* struct timeval: el primer camp s n segons,
40     * el segon camp s n microsegons */
41    struct timeval tv = { 5, 0 };
42    if (setsockopt (sockfd, SOL_SOCKET, SO_RCVTIMEO, &tv, sizeof (tv)) < 0)
43      perror ("SO_RCVTIMEO");
44
45    char msg[] = "Hello, world!";
46    write (sockfd, msg, sizeof (msg));
47
48    char buff[1500];
49    if (read (sockfd, buff, 1500) < 0)
50      perror ("read");
51    else
52      printf ("Ok\n");
53  }
```

12.3.6 Opció SO_REUSEADDR

Aquesta opció permet relaxar algunes restriccions que s'apliquen sobre la funció `bind`; és a dir, amb aquesta opció s'evita que `bind` falli en segons quines circumstàncies (el lector interessat por consultar les pàgines 210–213 de Stevens et al. [2004] per a més detalls). Amb aquesta opció es permet obrir un port que pertany a una connexió en l'estat TIME_WAIT; també es pot obrir un port que pertany a una connexió existent:

1. S'inicia un servidor que escolta un determinat port.

2. Arriba una nova connexió, i el servidor crea un fill per atendre la connexió.
3. El servidor (el procés pare) finalitza, però el fill segueix servint la connexió.
4. El servidor es torna a iniciar.

Al reiniciar el servidor, es trobarà que la crida a `bind` fallarà (*Address already in use*). Per tal que això no succeeixi, cal activar l'opció `SO_REUSEADDR`.

Per a aquesta opció, `optval` apunta a un enter que val 1 si es vol activar l'opció, i zero si es vol desactivar.

Exemple Com a exemple, es presenta una versió modificada de l'exemple de l'apartat 7.3.1, en què s'activa l'opció `SO_REUSEADDR` abans de cridar `bind`.

```
1  #include <arpa/inet.h>
2  #include <netinet/in.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <sys/socket.h>
7
8  int
9  main (int argc, char *argv[])
10 {
11   if (argc < 2)
12     {
13       printf ("Falta indicar el port\n");
14       return -1;
15     }
16
17   int sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
18   if (sockfd < 0)
19     {
20       perror ("socket");
21       return -1;
22     }
23
24   struct sockaddr_in s_addr;
25   memset (&s_addr, 0, sizeof (s_addr));
26   s_addr.sin_family = AF_INET;
27   s_addr.sin_port = htons (atoi (argv[1]));
28   s_addr.sin_addr.s_addr = INADDR_ANY;
29
30   int opt = 1;
31   setsockopt (sockfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof (opt));
32
33   if (bind (sockfd, (void *) &s_addr, sizeof (s_addr)) < 0)
34     {
35       perror ("bind");
```

```
36        return -1;
37      }
38
39    listen (sockfd, 3);
40  }
```

Nota La pàgina manual de Linux (`socket(7)`) fa el següent apunt:

> Linux will only allow port reuse with the `SO_REUSEADDR` option when this option was set both in the previous program that performed a `bind(2)` to the port and in the program that wants to reuse the port. This differs from some implementations (e.g., FreeBSD) where only the later program needs to set the `SO_REUSEADDR` option. Typically this difference is invisible, since, for example, a server program is designed to always set this option.

La qual cosa ve a dir que, en Linux, és altament recomanable que qualsevol aplicació que obri un port TCP activi aquesta opció.

12.3.7 Opció `SO_BINDTODEVICE`

Aquesta opció permet que l'aplicació se salti la taula de routing del sistema operatiu. Exactament, el que fa és que tots els segments d'un determinat socket surtin per una interfície en concret. Cal tenir en compte que aquesta opció només la pot activar un procés amb permisos de superusuari.

Convé fer algun aclariment sobre les diferències entre `bind` i aquesta opció de socket. La funció `bind` *lliga* adreces IP, no interfícies. És a dir, per a un servidor permet especificar quina adreça IP destí han de tenir els datagrames per tal que s'entreguin al socket, i els datagrames que surtin per aquest socket portaran, com a adreça IP origen, l'especificada amb la funció `bind`. Ara bé, quina interfície farà servir el sistema operatiu per a entregar el paquet, és una cosa en la que `bind` no hi intervé. Per exemple, es pot fer servir l'adreça IP de la interfície `eth0`, però el sistema operatiu pot decidir (per les taules de routing) que els datagrames han de sortir per la interfície `eth1`.

L'opció `SO_BINDTODEVICE` força que els paquets hagin de sortir per una interfície determinada, independentment del que digui la taula de routing. A més a més, els paquets entrants també han d'entrar per la interfície en qüestió; un paquet que porti l'adreça destí correcta però no entri per aquella interfície no s'entregarà al socket.

Per a aquesta opció, `optval` apunta a un objecte de tipus `struct ifreq`. El camp `ifr_name` contindrà el nom de la interfície en qüestió.

Exemple


```
#include <net/if.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>

void set_device (int sock_fd, char if_name[IFNAMSIZ]);

int
main (int argc, char *argv[])
{
  if (argc < 2)
    {
      printf ("Falta indicar la interf cie\n");
      return -1;
    }

  int sock_fd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
  set_device (sock_fd, argv[1]);
}

void
set_device (int sock_fd, char if_name[IFNAMSIZ])
{
  struct ifreq ifr;
  strncpy (ifr.ifr_name, if_name, IFNAMSIZ);

  printf ("Iface: %s\n", if_name);
  if (setsockopt (sock_fd, SOL_SOCKET, SO_BINDTODEVICE, &ifr, sizeof (ifr)) <
      0)
    perror ("setsockopt");
}
```

Exemple de crida (suposant que l'executable es diu `a.out`):

```
$ ./a.out eth0
Iface: eth0
setsockopt: Operation not permitted
$ sudo ./a.out eth0
Iface: eth0
```

12.3.8 Opció SO_KEEPALIVE

Aquesta opció afecta els sockets de connexions TCP. Per defecte, el protocol TCP no té cap mecanisme per tal de controlar si una determinada connexió segueix activa. És a dir, si cap de les dues parts envia dades, no hi ha cap transmissió de paquets, i per tant si hi hagués algun error a la xarxa, no es detectaria (per exemple, que algun router caigués, etc).

Activant aquesta opció de socket, quan una connexió està dues hores inactiva (és a dir, sense intercanviar dades), s'envia un segment *keep-alive*, que l'altra banda ha de respondre.

Exemple Aquest exemple és similar a l'exemple de la funció `connect` ofert al capítol 7, però activant l'opció `SO_KEEPALIVE`.

```
1  #include <arpa/inet.h>
2  #include <netinet/in.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <sys/socket.h>
7
8  int
9  main (int argc, char *argv[])
10 {
11   if (argc < 3)
12     {
13       printf ("Falten par metres (IP i port dest )\n");
14       return -1;
15     }
16
17   int sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
18   if (sockfd < 0)
19     {
20       perror ("socket");
21       return -1;
22     }
23
24   struct sockaddr_in s_addr;
25   memset (&s_addr, 0, sizeof (s_addr));
26   s_addr.sin_family = AF_INET;
27   s_addr.sin_port = htons (atoi (argv[2]));
28   if (inet_aton (argv[1], &s_addr.sin_addr) == 0)
29     {
30       printf ("%s no  s una adre a IP v lida\n", argv[1]);
31       return -1;
32     }
33
34   if (connect (sockfd, (void *) &s_addr, sizeof (s_addr)) < 0)
35     {
36       perror ("connect");
37       return -1;
38     }
39
40   int opt = 1;
41   setsockopt (sockfd, SOL_SOCKET, SO_KEEPALIVE, &opt, sizeof (opt));
42 }
```


12.3.9 Opció SO_ACCEPTCONN

Aquesta opció (que només és de lectura) permet saber si un socket accepta connexions o no, és a dir, si s'ha cridat `listen` amb aquest socket.

Exemple Aquest exemple és similar a l'exemple de la funció `accept` ofert al capítol 7, però s'hi afegeixen uns missatges que expliciten quin dels dos file descriptors accepta connexions.

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>

static int do_tcp_passive_open (const char *port);

int
main (int argc, char *argv[])
{
  if (argc < 2)
    {
      printf ("Falta indicar el port\n");
      return -1;
    }

  int sockfd = do_tcp_passive_open (argv[1]);

  struct sockaddr_in s_addr;
  socklen_t len = sizeof (s_addr);

  int newsock = accept (sockfd, (void *) &s_addr, &len);
  if (newsock < 0)
    {
      perror ("accept");
      return -1;
    }

  printf ("Nova connexi  procedent de %s:%hu\n", inet_ntoa (s_addr.sin_addr),
          ntohs (s_addr.sin_port));

  int v1, v2;
  len = sizeof (int);

  getsockopt (sockfd, SOL_SOCKET, SO_ACCEPTCONN, &v1, &len);
  getsockopt (newsock, SOL_SOCKET, SO_ACCEPTCONN, &v2, &len);

  if (v1)
```

```
41      printf ("sockfd accepta noves connexions\n");
42    else
43      printf ("sockfd no accepta noves connexions\n");
44    if (v2)
45      printf ("newsock accepta noves connexions\n");
46    else
47      printf ("newsock no accepta noves connexions\n");
48  }
49
50  static int
51  do_tcp_passive_open (const char *port)
52  {
53    int sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
54    if (sockfd < 0)
55      {
56        perror ("socket");
57        exit (-1);
58      }
59
60    struct sockaddr_in s_addr;
61    memset (&s_addr, 0, sizeof (s_addr));
62    s_addr.sin_family = AF_INET;
63    s_addr.sin_port = htons (atoi (port));
64    s_addr.sin_addr.s_addr = INADDR_ANY;
65
66    if (bind (sockfd, (void *) &s_addr, sizeof (s_addr)) < 0)
67      {
68        perror ("bind");
69        exit (-1);
70      }
71
72    listen (sockfd, 3);
73    return sockfd;
74  }
```

12.3.10 Opció TCP_MAXSEG

Amb aquesta opció es pot consultar (i fins i tot també modificar, **dins d'uns límits**) el paràmetre MSS del socket, és a dir, el *Maximum Segment Size* d'una connexió TCP. Aquest paràmetre indica quina quantitat màxima de *payload* pot portar un segment TCP. És molt desaconsellable modificar-lo, ja que depèn de l'MTU (*Maximum Transfer Unit*, que depèn del protocol de nivell 2, com ara Ethernet) i de les capçaleres dels protocols de nivell 3 (IP) i 4 (TCP). Per exemple, Ethernet té un MTU de 1500 bytes, ja que una trama d'aquest protocol pot portar, com a màxim, 1500 bytes de *payload*. La capçalera IP ocupa, com a mínim, 20 bytes, i la TCP 20, però com que generalment hi ha opcions, a la pràctica sovint n'ocupa 32. Això vol dir que un MSS típic és de $1500 - 20 - 32 = 1448$ bytes.

Exemple Com a exemple es presenta una aplicació que obre un port TCP (especificat per línia de comandes), i espera una connexió. Quan arriba la connexió en mostra la procedència i també el paràmetre MSS. Es proposa, com a exercici per al lector, modificar aquest exemple per tal que l'exemple de l'apartat 12.3.5 el pugui fer servir com a servidor.

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>

int
main (int argc, char *argv[])
{
  if (argc < 2)
    {
      fprintf (stderr, "Falta especificar el port\n");
      return -1;
    }

  int sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (sockfd < 0)
    {
      perror ("socket");
      return -1;
    }

  struct sockaddr_in addr = { AF_INET, 0, {0}, {0} };
  addr.sin_port = htons (atoi (argv[1]));
  if (addr.sin_port == 0)
    {
      fprintf (stderr, "%s no  s un valor v lid de port\n", argv[1]);
      return -1;
    }

  if (bind (sockfd, (void *) &addr, sizeof (addr)) < 0)
    {
      perror ("bind");
      return -1;
    }

  struct sockaddr_in src = { AF_INET, 0, {0}, {0} };
  socklen_t len = sizeof (src);

  printf ("Esperant...\n");
  int newfd = accept (sockfd, (void *) &src, &len);
  if (newfd < 0)
```

```
44      {
45        perror ("accept");
46        return -1;
47      }
48    printf ("Nova connexi  de %s:%hu\n", inet_ntoa (src.sin_addr),
49            ntohs (src.sin_port));
50
51    /* NOTE: cal haver incl s netinet/tcp.h perqu  estigui definida
52     * la macro ("constant") TCP_MAXSEG */
53    int mss;
54    len = sizeof (int);
55    if (getsockopt (newfd, IPPROTO_TCP, TCP_MAXSEG, &mss, &len) < 0)
56      perror ("TCP_MAXSEG");
57    else
58      printf ("Mida de l'MSS: %d\n", mss);
59  }
```

12.3.11 Opció TCP_NODELAY

Aquesta opció activa o desactiva l'algorisme Nagle (vegeu [Stevens, 1994, apartat 19.4] i [Wright and Stevens, 1995, pàgines 858–859]). Quan aquesta opció està activada, l'algorisme Nagle està desactivat; i vice-versa.

La finalitat de l'algorisme Nagle és reduir el nombre de segments petits, entenent per segment petit aquell que és inferior a l'MSS. El funcionament de l'algorisme és el següent: si per un socket s'han enviat dades que encara no s'han confirmat (és a dir, per a les quals no s'ha rebut el corresponent segment ACK), i l'usuari vol enviar més dades, no s'enviarà cap segment petit. El protocol TCP intenta enviar, sempre que sigui possible, segments "sencers" (és a dir, aprofitant tot l'MSS), el que intenta l'algorisme Nagle és que hi hagi diversos segments petits pendents de confirmar. Per a una comparació de com actua el protocol amb l'algorisme activat i desactivat, vegeu [Stevens et al., 2004, pàgines 220 i 221].

Exemple Aquest exemple és similar a l'exemple de la funció `connect` ofert al capítol 7, però activant l'opció `TCP_NODELAY`.

```
1  #include <arpa/inet.h>
2  #include <netinet/in.h>
3  #include <netinet/tcp.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <sys/socket.h>
8
9  int
10 main (int argc, char *argv[])
11 {
12   if (argc < 3)
```



```
13      {
14        printf ("Falten par metres (IP i port dest )\n");
15        return -1;
16      }
17
18    int sockfd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
19    if (sockfd < 0)
20      {
21        perror ("socket");
22        return -1;
23      }
24
25    struct sockaddr_in s_addr;
26    memset (&s_addr, 0, sizeof (s_addr));
27    s_addr.sin_family = AF_INET;
28    s_addr.sin_port = htons (atoi (argv[2]));
29    if (inet_aton (argv[1], &s_addr.sin_addr) == 0)
30      {
31        printf ("%s no  s una adre a IP v lida\n", argv[1]);
32        return -1;
33      }
34
35    if (connect (sockfd, (void *) &s_addr, sizeof (s_addr)) < 0)
36      {
37        perror ("connect");
38        return -1;
39      }
40
41    int opt = 1;
42    setsockopt (sockfd, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof (opt));
43  }
```

12.4 Protocol UDP

12.4.1 Introducció

El protocol UDP presenta una sèrie de diferències fonamentals amb UDP:

1. No és orientat a connexió. La funció `connect` es pot cridar, però amb un sentit totalment diferent del de TCP.
2. Un socket UDP pot enviar a diversos destinataris, i pot rebre de diversos emissors.
3. En UDP no hi ha control de flux, ni cap garantia sobre si les dades arriben al seu destí.
4. En UDP s'envien *missatges* (que no poden excedir els 65535 bytes), no un flux de bytes.

Totes aquestes diferències tenen el seu reflex a l'hora de programar amb sockets UDP, com s'anirà veient en aquest apartat.

12.4.2 Creació del socket i funció *bind*

Crear un socket en UDP és pràcticament idèntic a TCP:

```
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>

int
main (void)
{
  int sockfd = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if (sockfd < 0)
    {
      perror ("socket");
      return -1;
    }
}
```

En UDP també hi ha el concepte d'obrir port, és a dir, establir una adreça IP i/o port per tal de poder rebre missatges. Si no es crida la funció `bind`, el sistema operatiu assigna adreça i port el primer cop que s'envia un missatge a algú. En UDP la funció `bind` funciona exactament igual que en TCP — però en UDP no hi ha estat TIME_WAIT.

El següent exemple crea un socket UDP i li assigna el port que s'especifica per línia de comandes:

```
#include <arpa/inet.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>

int
main (int argc, char *argv[])
{
  if (argc < 2)
    {
      printf ("Falta indicar el port\n");
      return -1;
    }

  int sockfd = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

```
18    if (sockfd < 0)
19      {
20        perror ("socket");
21        return -1;
22      }
23
24    struct sockaddr_in s_addr;
25    memset (&s_addr, 0, sizeof (s_addr));
26    s_addr.sin_family = AF_INET;
27    s_addr.sin_port = htons (atoi (argv[1]));
28    s_addr.sin_addr.s_addr = INADDR_ANY;
29
30    if (bind (sockfd, (void *) &s_addr, sizeof (s_addr)) < 0)
31      {
32        perror ("bind");
33        return -1;
34      }
35  }
```

Tal com es pot veure, no es crida la funció `listen`.

12.4.3 Funcions *sendto* i *recvfrom*

Per a enviar i rebre dades en UDP, les funcions `read`, `write`, `send` i `recv` no són apropiades, ja que no permeten especificar el destí (ni permeten saber l'emissor) – però vegeu l'apartat 12.4.5. Per tant, hi ha dues noves funcions:

```
#include <sys/socket.h>

ssize_t sendto (int sockfd, const void *buff, size_t nbytes,
                int flags, const struct sockaddr *to,
                socklen_t addrlen);

ssize_t recvfrom (int sockfd, void *buff, size_t nbytes,
                  int flags, struct sockaddr *from,
                  socklen_t *addrlen);
```

- La funció `sendto` envia dades al socket especificat pel file descriptor `sockfd`. Les dades estan a la memòria apuntada per `buff`, i s'envien `nbytes` bytes, al destinatari especificat per `to`. El paràmetre `addrlen` és la mida de l'adreça apuntada per `to`, i `flags` permet modificar alguns aspectes de com s'ha d'efectuar l'enviament. La funció retorna el número de bytes enviats.
- La funció `recvfrom` rep dades pel socket especificat pel file descriptor `sockfd`. Les dades es guarden a la memòria apuntada per `buff`, que pot guardar fins a `nbytes` bytes (que és el màxim de bytes que es llegiran. L'adreça del destinatari es guarda a la memòria apuntada per `from`, i `addrlen` al principi indica

la mida de la memòria apuntada per `from`. El paràmetre `flags` permet modificar alguns aspectes de com s'ha d'efectuar l'enviament. La funció retorna el número de bytes llegits.

Si no estem interessats en saber qui ens ha enviat el missatge, podem passar `NULL` als paràmetres `from` i `addrlen`.

Aquestes dues funcions també es podrien emprar per a TCP, però tenint en compte que TCP és un protocol orientat a connexió, no hi ha cap raó per a fer-ho. Per cert, en UDP es permet enviar un missatge buit; en aquest cas, s'enviarà un datagrama IP que contindrà només les capçaleres IP i UDP.

Què passa si el missatge rebut és més gran que la mida especificada per `nbytes`? En aquest cas, només es llegeixen `nbytes` bytes, i la resta es perden.

Exemple Aquest exemple obre un port UDP, i espera que algú li enviï un missatge. Quan el rep, el torna a enviar al seu destinatari.

```
1  #include <arpa/inet.h>
2  #include <netinet/in.h>
3  #include <stdint.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <sys/socket.h>
8  #include <unistd.h>
9
10 static int open_udp_port (uint16_t port);
11
12 int
13 main (int argc, char *argv[])
14 {
15   if (argc < 2)
16     {
17       printf ("Falta indicar el port\n");
18       return -1;
19     }
20
21   int udp_fd = open_udp_port (atoi (argv[1]));
22   if (udp_fd < 0)
23     return -1;
24
25   char buff[64 * 1024];
26   struct sockaddr_in addr = { AF_INET, 0, {0}, {0} };
27   socklen_t len = sizeof (addr);
28
29   ssize_t sst = recvfrom (udp_fd, buff, 64 * 1024, 0, (void *) &addr, &len);
30   if (sst < 0)
31     perror ("recvfrom");
```



```
32    else
33      printf ("Rebuts %lu bytes de %s:%hu\n", sst, inet_ntoa (addr.sin_addr),
34              ntohs (addr.sin_port));
35
36    sendto (udp_fd, buff, sst, 0, (void *) &addr, len);
37  }
38
39  static int
40  open_udp_port (uint16_t port)
41  {
42    int sockfd = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
43    if (sockfd < 0)
44      {
45        perror ("socket");
46        return -1;
47      }
48
49    struct sockaddr_in s_addr;
50    memset (&s_addr, 0, sizeof (s_addr));
51    s_addr.sin_family = AF_INET;
52    s_addr.sin_port = htons (port);
53    s_addr.sin_addr.s_addr = INADDR_ANY;
54
55    if (bind (sockfd, (void *) &s_addr, sizeof (s_addr)) < 0)
56      {
57        perror ("bind");
58        close (sockfd);
59        return -1;
60      }
61
62    return sockfd;
63  }
```

Flags Els valors que pot prendre el paràmetre `flags` s'explicaran a l'apartat següent.

12.4.4 Funcions *sendmsg* i *recvmsg*

Es tracta de les funcions més generals per a enviar i rebre dades. Estan declarades de la següent manera:

```
#include <sys/socket.h>
#include <sys/uio.h>

ssize_t sendmsg (int sockfd, struct msghdr *msg, int flags);
ssize_t recvmsg (int sockfd, struct msghdr *msg, int flags);
```

El tipus `struct msghdr` està definit de la següent manera:

```
struct msghdr
{
  void *msg_name;
  socklen_t msg_namelen;
  struct iovec *msg_iov;
  int msg_iovlen;
  void *msg_control;
  int msg_controllen;
  int flags;
};
```

Els camps `msg_control` i `msg_controllen` no els explicarem en aquest llibre.

- El camp `msg_name` és equivalent a `to` i `from` de les funcions `sendto` i `recvfrom`.
- El camp `msg_namelen` és equivalent a `addrlen` de les funcions `sendto` i `recvfrom`. En aquestes dues últimes funcions, a la segona era un punter i a la primera no. En les funcions `sendmsg` i `recvmsg` no cal que ho sigui, ja que tota l'estructura és un punter en els dos casos.
- El camp `msg_iov` és equivalent al paràmetre `iov` de les funcions `readv` i `writev`.
- El camp `msg_iovlen` és equivalent al paràmetre `iovlen` de les funcions `readv` i `writev`.
- El camp `flags` són les `flags` que retorna la funció `recvfrom`. És a dir, en aquest cas, el programador especifica flags a través del paràmetre `flags` de la funció `recvfrom`, i el sistema operatiu en pot activar a través del camp `msg_flags` del paràmetre `msg`. La funció `sendmsg` no fa servir aquest camp.

La funció `sendmsg` retorna la quantitat de bytes que ha enviat; la funció `recvmsg` retorna la quantitat de bytes llegits.

Exemple El següent exemple és el mateix que el de l'apartat anteior, però usant `recvmsg` en lloc de `recvfrom` i `sendmsg` en lloc de `sendto`.

```
1  #include <arpa/inet.h>
2  #include <netinet/in.h>
3  #include <stdint.h>
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <string.h>
7  #include <sys/socket.h>
8  #include <unistd.h>
9
10 static int open_udp_port (uint16_t port);
```



```
11
12  int
13  main (int argc, char *argv[])
14  {
15    if (argc < 2)
16      {
17        printf ("Falta indicar el port\n");
18        return -1;
19      }
20
21    int udp_fd = open_udp_port (atoi (argv[1]));
22    if (udp_fd < 0)
23      return -1;
24
25    char buff[64 * 1024];
26    struct msghdr msg;
27    memset (&msg, 0, sizeof (msg));
28    struct sockaddr_in addr = { AF_INET, 0, {0}, {0} };
29    msg.msg_name = &addr;
30    msg.msg_namelen = sizeof (addr);
31    struct iovec iov = { buff, 64 * 1024 };
32    msg.msg_iov = &iov;
33    msg.msg_iovlen = 1;
34    msg.msg_control = NULL;
35    msg.msg_controllen = 0;
36
37    ssize_t sst = recvmsg (udp_fd, &msg, 0);
38    if (sst < 0)
39      perror ("recvfrom");
40    else
41      printf ("Rebuts %lu bytes de %s:%hu\n", sst, inet_ntoa (addr.sin_addr),
42              ntohs (addr.sin_port));
43
44    iov.iov_len = sst;
45    sendmsg (udp_fd, &msg, 0);
46  }
47
48  static int
49  open_udp_port (uint16_t port)
50  {
51    int sockfd = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
52    if (sockfd < 0)
53      {
54        perror ("socket");
55        return -1;
56      }
57
58    struct sockaddr_in s_addr;
59    memset (&s_addr, 0, sizeof (s_addr));
```

```
60    s_addr.sin_family = AF_INET;
61    s_addr.sin_port = htons (port);
62    s_addr.sin_addr.s_addr = INADDR_ANY;
63
64    if (bind (sockfd, (void *) &s_addr, sizeof (s_addr)) < 0)
65      {
66        perror ("bind");
67        close (sockfd);
68        return -1;
69      }
70
71    return sockfd;
72  }
```

Flags Hi ha diversos flags que es poden activar per a `sendto`, `sendmsg`, `recvfrom` i `recvmsg`. Els més interessants són els següents:

`MSG_DONTWAIT` Fa que l'operació sigui no bloquejant, és a dir, com si el socket estigués en mode no bloquejant. Vegeu l'apartat 12.5 per a més detalls sobre aquest mode. Vàlid per a `sendmsg` i `recvmsg`.

`MSG_MORE` En TCP, fa que l'operació actuï com si el socket tingués activada l'opció `TCP_CORK` (vegeu l'apartat **??**. Per a UDP aquesta opció també és possible, i indica que el missatge encara no s'ha acabat, és a dir, que es formarà amb diverses crides a `sendmsg`; el missatge s'acabarà amb la crida que no tingui activat aquest flag. Vàlid només per a `sendmsg`.

`MSG_NOSIGNAL` Evita que es rebi el senyal `SIGPIPE` si, en TCP, no hi ha ningú que escolti la connexió. Vàlid només per a `sendmsg`.

`MSG_PEEK` Fa que `recvfrom` no elimini les dades del buffer del sistema operatiu; és a dir, la següent lectura retornarà les mateixes dades. No és vàlid per a `sendmsg`.

`MSG_WAITALL` Per a TCP, indica que la funció `recvfrom` s'ha d'esperar que arribin tots els bytes indicats. No és vàlid per a `sendmsg`.

`MSG_TRUNC` Quan s'especifica aquest flag com a paràmetre de `recvmsg`, s'indica que volem que `recvmsg` retorni (com a valor de retorn) la mida total del missatge, encara que el trunqui. Quan aquest flag apareix al camp `msg_flags`, llavors vol dir que `recvmsg` ens està dient que el missatge era massa gran i per tant l'ha truncat.

12.4.5 Funció *connect* per a UDP

La funció `connect`, per a UDP, es crida exactament igual que per a TCP, però té un significat completament diferent. En UDP significa que el destinatari dels segments

serà sempre l'adreça especificada al cridar `connect`, i s'ignorarà les que es passen a les funcions `sendto` i `sendmsg` (de fet, s'hauria de passar `NULL` com a arguments a `to` i `from`). A més a més, només podrem rebre dades del destinatari especificat. D'aquesta manera, no cal emprar les funcions vistes als dos apartats anteriors, sinó que ja podem usar `read` i `write` (o `recv` i `send`).

Es pot cridar la funció `connect` diverses vegades, per tal de canviar de destinatari (i receptor). També és possible *anul·lar* l'efecte de la crida a `connect`, de manera que sigui com si mai s'hagués cridat, de la següent manera:

```
1  #include <arpa/inet.h>
2  #include <sys/socket.h>
3
4  void
5  foo (int sockfd)
6  {
7    struct sockaddr_in addr = { AF_UNSPEC, 0, {0}, {0} };
8
9    connect (sockfd, (void *) &addr, sizeof (addr));
10 }
```

12.5 Sockets no bloquejants

12.5.1 Introducció

Per defecte, les operacions sobre sockets són bloquejants. Això vol dir que les funcions relacionades amb sockets no retornen fins que no s'ha completat la seva tasca. Quan un socket es troba en mode no bloquejant, canvia el comportament d'algunes funcions relacionades amb sockets:

1. Funcions d'entrada (com ara `read`, `readv`, `recv`, `recvfrom` i `recvmsg`): si no hi ha dades al buffer d'entrada, el procés queda bloquejat. Per a un socket TCP, aquestes funcions s'esperen que arribin dades, ja sigui un sol byte o més. Si el que es vol és esperar que arribi una determinada quantitat de dades, llavors cal especificar el flag `MSG_WAITALL` a les funcions `recv`, `recvfrom` o `recvmsg`. Per als sockets UDP, si el buffer està buit, les funcions s'esperen que arribi un segment UDP.

   Si el socket no és bloquejant, llavors el procés no queda bloquejat. Si hi ha dades, es retornen les dades que hi hagi, segons el que ja s'ha explicat; però si no hi ha dades, llavors les funcions retornen -1 amb el codi d'error `EWOULDBLOCK`.

2. Funcions de sortida (com ara `write`, `writev`, `send`, `sendto` i `sendmsg`): si no hi ha espai al buffer de sortida, el procés queda bloquejat fins que hi hagi prou espai (si hi ha prou espai es copien les dades i les funcions retornen).

   Si el socket no és bloquejant, llavors el procés no queda mai bloquejat. Si no hi ha prou espai al buffer, es copien els bytes que hi caben (si almenys n'hi

cap un) i les funcions retornen la quantitat de bytes que s'han copiar al buffer de sortida; si no n'hi cap ni un, llavors les funcions retornen -1 amb el codi d'error `EWOULDBLOCK`.

3. Funció `accept`: per defecte, aquesta funció bloqueja el procés fins que arriba una connexió. Per a sockets no bloquejants, si al moment de cridar `accept` no ha arribat cap connexió llavors retorna -1 amb el codi d'error `EWOULDBLOCK`.

4. Funció `connect`: per defecte, aquesta funció bloqueja el procés fins que es completa l'establiment de la connexió (per a TCP, fins que rep el segment `SYN+ACK` per part del servidor). Per a sockets no bloquejants, `connect` inicia l'establiment però no s'espera que es completi. Cal tenir en compte que la funció `connect` retorna -1 però amb un codi d'error diferent: `EINPROGRESS`, per indicar que es tracta d'una operació que s'està duent a terme i que, en algun moment futur, es completarà.

   Cal tenir en compte que pot ser que `connect` retorni immediatament fins i tot per a un socket no bloquejant, quan el client i el servidor s'executen a la mateixa màquina.

12.5.2 Creació d'un socket no bloquejant

Per defecte, tots els sockets que es creen són bloquejants. Un cop creat un socket, es pot canviar el seu mode de funcionament amb la funció `fcntl`, com s'il·lustra en l'exemple següent.

```
1  #include <fcntl.h>
2
3  void
4  socket_to_blocking (int fd)
5  {
6    int flags = fcntl (fd, F_GETFL, 0);
7    flags &= ~O_NONBLOCK;
8    fcntl (fd, F_SETFL, flags);
9  }
10
11 void
12 socket_to_nonblock (int fd)
13 {
14   int flags = fcntl (fd, F_GETFL, 0);
15   flags |= O_NONBLOCK;
16   fcntl (fd, F_SETFL, flags);
17 }
```

Des de la versió 2.6.27 del kernel de Linux, al mateix temps que es crea un socket es pot especificar que sigui no bloquejant:


```
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>

int
main (void)
{
  int sockfd = socket (AF_INET, SOCK_STREAM | SOCK_NONBLOCK, IPPROTO_TCP);
  if (sockfd < 0)
    {
      perror ("socket");
      return -1;
    }
}
```

12.5.3 Connexions no bloquejants

Certes aplicacions necessiten realitzar una certa quantitat de connexions en un determinat moment. Amb sockets bloquejants, l'aplicació farà les connexions una rere l'altra, i no intentarà cap connexió fins que l'anterior s'hagi completat (o s'hagi detectat un error). Per raons d'eficiència això pot ser perjudicial perquè podria consumir una gran quantitat de temps (el *timeout* d'una connexió TCP sol ser de 75 segons, i per al protocol DCCP sol ser d'uns dos minuts).

Amb sockets no bloquejants, aquest inconvenient queda eliminat, però la complexitat del programa augmenta: la funció `connect` retorna quan s'inicia l'establiment de la connexió, però com sabem que ja s'ha completat, o que s'ha produït un error? Cal recórrer a funcions com ara `select`.

Per a realitzar connexions no bloquejants cal seguir els següents passos:

1. Partim d'un socket que es troba en mode no bloquejant, ja sigui perquè s'ha creat d'aquesta manera o bé perquè se l'ha establert en aquest mode posteriorment.

2. Cridem la funció `connect` de la manera habitual, però comprovant el seu codi de retorn:

   - Si retorna 0, vol dir que la connexió s'ha establert correctament.
   - Si retorna -1 i `errno` val `EINPROGRESS`, llavors vol dir que `connect` ha iniciat l'establiment de la connexió, però encara no s'ha completat.
   - Si retorna -1 i `errno` no és `EINPROGRESS`, llavors vol dir que hi ha hagut algun error.

3. Per a aquells file descriptors corresponents a connexions en procés, es crida la funció `select`. Quan el file descriptor en qüestió és a punt per a escriptura, s'empra l'opció de socket `SO_ERROR` per saber si la connexió s'ha establert correctament o si hi ha hagut algun error.

Tot això es veu millor amb un exemple.

Exemple L'exemple consisteix en una aplicació que rep una adreça IP i una llista de ports TCP. L'aplicació inicia tantes connexions com ports s'especifiquen.

```
#include <arpa/inet.h>
#include <errno.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

int
main (int argc, char *argv[])
{
  /* We check the number of arguments */
  if (argc < 3)
    {
      printf ("Falten par metres (IP i port dest )\n");
      return -1;
    }

  /* We store the specified IP address in a variable */
  struct in_addr ip_addr;
  if (inet_aton (argv[1], &ip_addr) == 0)
    {
      printf ("%s no  s una adre a IP v lida\n", argv[1]);
      return -1;
    }

  /* For each specified port, we create a socket
   * and try to start a connection */
  int num_ports = argc - 2;
  int fds[num_ports];
  struct sockaddr_in addrs[num_ports];
  memset (fds, 0, sizeof (int) * num_ports);
  memset (addrs, 0, sizeof (struct sockaddr_in) * num_ports);
  int i;

  int num_conns = 0;
  for (i = 0; i < num_ports; i++)
    {
      /* Step number 1 */
      fds[i] = socket (AF_INET, SOCK_STREAM | SOCK_NONBLOCK, IPPROTO_TCP);
      if (fds[i] < 0)
        perror ("socket");

```

```
45        addrs[i].sin_family = AF_INET;
46        addrs[i].sin_addr = ip_addr;
47        addrs[i].sin_port = htons (atoi (argv[2 + i]));
48
49        /* Step 2 */
50        if (connect (fds[i], (void *) &addrs[i], sizeof (addrs[i])) < 0)
51          {
52            if (errno == EINPROGRESS)
53              {
54                /* The connection is in progress */
55                printf ("Started connection to %s:%hu\n",
56                        inet_ntoa (addrs[i].sin_addr),
57                        ntohs (addrs[i].sin_port));
58                num_conns++;
59              }
60            else
61              {
62                /* The connection could not be started */
63                fprintf (stderr,
64                         "Could not start connection to %s:%hu: "
65                         "%s (errno = %d)\n", inet_ntoa (addrs[i].sin_addr),
66                         ntohs (addrs[i].sin_port), strerror (errno), errno);
67                close (fds[i]);
68                fds[i] = -1;
69              }
70          }
71        else
72          {
73            /* The connection has just been completed */
74            printf ("Connection instantly established with %s:%hu ",
75                    inet_ntoa (addrs[i].sin_addr), ntohs (addrs[i].sin_port));
76            struct sockaddr_in from = { AF_INET, 0, {0}, {0} };
77            socklen_t len = sizeof (from);
78            getsockname (fds[i], (void *) &from, &len);
79            printf ("from %s:%hu\n", inet_ntoa (from.sin_addr),
80                    ntohs (from.sin_port));
81            close (fds[i]);
82            fds[i] = -1;
83          }
84      }
85
86    /* Step 3 */
87    fd_set set;
88    FD_ZERO (&set);
89
90    /* For each pending connections, we await until completion of error */
91    printf ("Waiting for %d connections...\n", num_conns);
92    while (num_conns)
93      {
```

```
94        for (i = 0; i < num_ports; i++)
95          if (fds[i] > 0)
96            FD_SET (fds[i], &set);
97
98        switch (select (num_ports + 16, NULL, &set, NULL, NULL))
99          {
100         case -1:
101           perror ("select");
102           return -1;
103
104         case 0:
105           continue;
106         }
107
108       for (i = 0; i < num_ports; i++)
109         if (fds[i] > 0 && FD_ISSET (fds[i], &set))
110           {
111             int the_error;
112             socklen_t len = sizeof (int);
113             getsockopt (fds[i], SOL_SOCKET, SO_ERROR, &the_error, &len);
114
115             if (the_error == 0)
116               {
117                 /* The connection has completed */
118                 printf ("Connection successfully established with %s:%hu ",
119                         inet_ntoa (addrs[i].sin_addr),
120                         ntohs (addrs[i].sin_port));
121                 struct sockaddr_in from = { AF_INET, 0, {0}, {0} };
122                 len = sizeof (from);
123                 getsockname (fds[i], (void *) &from, &len);
124                 printf ("from %s:%hu\n", inet_ntoa (from.sin_addr),
125                         ntohs (from.sin_port));
126                 close (fds[i]);
127                 fds[i] = -1;
128               }
129             else
130               {
131                 /* There was an error */
132                 fprintf (stderr,
133                          "Could not connect to %s:%hu: %s (errno = %d)\n",
134                          inet_ntoa (addrs[i].sin_addr),
135                          ntohs (addrs[i].sin_port), strerror (errno), errno);
136                 close (fds[i]);
137                 fds[i] = -1;
138               }
139             num_conns--;
140           }
141     }
142 }
```


12.6 Funció *ioctl* per a sockets

La funció `ioctl` és una funció genèrica per a operacions d'entrada i sortida, com una espècie de comodí o "navalla suïssa". Pel que fa a sockets, hi ha dues operacions força interessanta, que es presenten a continuació. (La pàgina manual de `socket(7)` ofereix algunes altres operacions que es poden realitzar amb `ioctl`).

12.6.1 FIONREAD

Aquesta operació retorna la quantitat de bytes que estan pendents de llegir en un socket.

Exemple

```
#include <sys/ioctl.h>

int
bytes_to_read (int fd)
{
  int nbytes;
  ioctl (fd, FIONREAD, &nbytes);
  return nbytes;
}
```

12.6.2 TIOCOUTQ

Aquesta operació retorna la quantitat de bytes que hi ha a la cua de transmissió d'un socket.

Exemple

```
#include <sys/ioctl.h>

int
bytes_to_write (int fd)
{
  int nbytes;
  ioctl (fd, TIOCOUTQ, &nbytes);
  return nbytes;
}
```

Capítol 13

Sockets Ethernet i operacions sobre interfícies de xarxa

13.1 Introducció

Tot i que som plenament conscients que, per a les pràctiques de l'assignatura, aquest capítol és virtualment innecessari, hem considerat adequat oferir-lo per mostrar, ni que sigui a un nivell il·lustratiu — o almenys no tan profund com la resta de capítols — unes eines relacionades amb sockets que poden ser molt interessants per a la realització de segons quines aplicacions, amb la qual cosa el lector pot adonar-se de fins a quin punt són flexibles els sockets com a eina per a treballar amb tot el que té a veure amb xarxes.

13.2 Sockets Ethernet

Els sockets d'aquesta família permeten accedir al més baix nivell; són útils per a la confecció d'*sniffers* tant passius (com ara *tcpdump* i *wireshark*) com actius (programes que permeten injectar paquets fets a mà a la xarxa).

13.2.1 Creació del socket

Els sockets de la família `AF_PACKET` permeten dues menes de tipus:

`SOCK_RAW` Tota la trama Ethernet es passa a l'aplicació. Quan es llegeix del socket, la capçalera Ethernet també s'entrega a l'aplicació. Quan s'escriu al socket, cal passar tota la capçalera Ethernet.

`SOCK_DGRAM` Només es passa a l'aplicació el *payload* de la trama Ethernet, i per tant al llegir del socket la capçalera Ethernet no s'entrega a l'aplicació. L'usuari pot saber les adreces MAC origen i destí a través d'un objecte de tipus `struct sockaddr_ll`. A l'escriure al socket no cal passar la capçalera Ethernet, el sistema operatiu la construeix a partir de les adreces que s'especifiquen amb un objecte de tipus `struct sockaddr_ll`.

L'últim argument que rep la funció `socket` pot prendre una gran varietat de valors, i serveix per a filtrar els paquets que l'aplicació vol rebre. Els dos valors més típics són:

`ETH_P_ALL` Es passen a l'aplicació totes les trames Ethernet, independentment del protocol de nivell superior.

`ETH_P_IP` Només es passen a l'aplicació les trames Ethernet que contenen datagrames IP.

Cal tenir en compte que, per defecte, s'escolten totes les interfícies Ethernet. Si es vol escoltar únicament una interfície, no serveix emprar l'opció `SO_BINDTODEVICE`, sinó que cal servir-se de la funció `bind`. A l'apartat 13.2.4 es presenta un exemple complet on, entre d'altres coses, es mostra com obrir un socket de nivell 2.

A la resta d'apartats següents, i per tal de simplificar i no voler abarcar excessiva matèria, se suposarà que el socket s'ha creat de tal manera que l'usuari no rep capçaleres Ethernet al llegir ni les ha de compondre a l'escriure, i que les trames Ethernet que s'entreguen a l'aplicació són les que contenen datagrames IP.

13.2.2 Lectura de trames

Per a llegir trames en aquesta mena de sockets s'empra la funció `recvfrom`:

```
ssize_t recvfrom (int sockfd, void *buff, size_t nbytes,
                  int flags, struct sockaddr *from,
                  socklen_t *addrlen);
```

`sockfd` File descriptor del socket de tipus `AF_PACKET`.

`buff` Punter al buffer on es guardarà la trama llegida.

`nbytes` Mida del buffer.

`flags` Típicament valdrà zero.

`from` Punter a un objecte de tipus `struct sockaddr`, que serà realment un punter a objecte de tipus `struct sockaddr_ll`.

`addrlen` Punter a una variable de tipus enter, que contindrà (abans de la crida) la mida de l'objecte `struct sockaddr_ll`.

La funció retorna la quantitat de bytes llegits. Al mateix temps, l'objecte apuntat per `from` conté informació relativa a la trama llegida:

- El camp `sll_family` valdrà `AF_PACKET`.
- El camp `sll_protocol` contindrà, en *network byte order* (o sigui, *big endian*), el protocol de nivell 3 de la trama Ethernet. Per exemple, si la trama Ethernet conté un datagrama IP, llavors valdrà `(ETH_P_IP)` (en *network byte order*).
- El camp `sll_ifindex` correspon a la interfície per on ha passat la trama, ja sigui perquè ha entrat per aquella interfície o perquè n'ha sortit.
- El camp `sll_hatype` contindrà el tipus de trama, és a dir, el tipus de protocol de nivell 2. El cas més típic és que valgui 1, valor que correspon a Ethernet. Aquest camp també està en *network byte order*.
- El camp `sll_pkttype` pot tenir un dels següents valors:

  `PACKET_HOST` La trama va dirigida al host local (és a dir, és una trama entrant).

  `PACKET_BROADCAST` La trama va dirigida a tots els equips de la xarxa.

  `PACKET_MULTICAST` La trama va dirigida a alguns equips de la xarxa (en Ethernet també hi ha adreces multicast).

  `PACKET_OTHERHOST` La trama va dirigida a un altre host de la xarxa, però no l'ha originat el host local. És a dir, s'ha rebut una trama en què ni l'origen ni el destí són el host local.

  `PACKET_OUTGOING` La trama l'ha originada el host local (és a dir, és una trama sortint).

- El camp `sll_halen` conté la mida, en bytes, de l'adreça de hardware; si es tracta d'una trama Ethernet, valdrà 6, ja que les trames Ethernet ocupen 48 bits (o sigui, 6 bytes).
- El camp `sll_addr` conté l'adreça MAC origen de la trama.

Si el socket és de tipus `SOCK_RAW`, al buffer s'hi inclourà la capçalera de nivell 2 (o sigui, la capçalera Ethernet). Si és de tipus `SOCK_DGRAM`, no s'inclourà tal capçalera.

13.2.3 Escriptura de trames

Per a escriure trames en aquesta mena de sockets s'empra la funció `sendto`. S'ha de passar un punter a un objecte de tipus `struct sockaddr_ll`, omplint els següents camps:

- El camp `sll_family` ha de valdre `AF_PACKET`.
- El camp `sll_halen` ha de valdre 6, ja que les adreces Ethernet ocupen 6 bytes (48 bits).
- El camp `sll_ifindex` ha de correspondre al de la interfície per la qual ha de sortir la trama Ethernet. Per tant, convé que es cridi `ioctl` per a consultar quin valor `ifindex` té la interfície que s'ha obert. A l'exemple que s'ofereix en aquest mateix apartat hi ha un exemple de com fer aquesta crida.
- El camp `sll_protocol` ha de valdre, en *network byte order* (o sigui, *big endian*), el protocol de nivell 3 de la trama Ethernet. Per exemple, si la trama Ethernet conté un datagrama IP, llavors ha de valdre `htons (ETH_P_IP)`.
- El camp `sll_addr` ha de contenir l'adreça MAC destí. En binari, evidentment. Per a la conversió d'adreces en format ASCII a format binari hi ha les funcions `ether_aton` i `ether_ntoa`:

  ```
  #include <netinet/ether.h>

  struct ether_addr *ether_aton (const char *asc);
  char *ether_ntoa (const struct ether_addr *addr);
  ```

Com a exemple, es presenta una funció que rep els següents paràmetres:

`fd` File descriptor corresponent a un socket de tipus `AF_PACKET`, degudament obert.

`ifname` Nom de la interfície per on es vol que surti la trama (per exemple, `eth0`).

`mac_desti` Adreça MAC destí de la trama Ethernet, passat com una cadena de caràcters.

`data` Punter a les dades que entraran dins del *payload* de la trama Ethernet (màxim 1500 bytes).

`size` Mida en bytes de la zona de memòria apuntada per `data`.

`proto` Protocol de nivell 3 (per exemple, `ETH_P_IP`).

```c
// sockets/level2_sendto.c
#include <arpa/inet.h>
#include <net/ethernet.h>
#include <net/if.h>
#include <netinet/ether.h>
#include <netpacket/packet.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

ssize_t
send_to_iface (const int fd, const char *ifname, const char *mac_desti,
               void *data, const size_t size, unsigned short proto)
{
  assert (ifname);
  assert (mac_desti);
  assert (data);

  // primer pas: obtenir el valor ifindex de la interf cie
  struct ifreq ifr;

  strncpy (ifr.ifr_name, ifname, IFNAMSIZ);
  if (ioctl (fd, SIOCGIFINDEX, &ifr) < 0)
    {
      perror ("ioctl");
      exit (EXIT_FAILURE);
    }

  // segon pas: convertir l'adre a MAC dest
  struct ether_addr *mac_addr = ether_aton (mac_desti);

  // tercer pas: omplir l'estructura sockaddr_ll
  struct sockaddr_ll desti;
  desti.sll_family = AF_PACKET;
  desti.sll_halen = 6;
  desti.sll_ifindex = ifr.ifr_ifindex;
  desti.sll_protocol = htons (proto);
  bcopy (mac_addr->ether_addr_octet, desti.sll_addr, 6);

  // finalment, fem sortir la trama
  return sendto (fd, data, size, 0, (void *) &desti, sizeof (desti));
}
```

13.2.4 Exemple: un sniffer

En aquest apartat es mostrarà un exemple complet que sintetitza el que s'ha explicat en aquest apartat sobre sockets de nivell 2. Es tracta d'una aplicació que mostrarà les trames que entren i/o surten per una determinada interfície de xarxa (o totes, si no se n'especifica cap). Les funcionalitats de l'aplicació exemple es limiten a escoltar únicament datagrames IP, i mostrar alguns paràmetres de la capçalera IP (i opcionalment de la capçalera TCP).

El programa rebrà les següents opcions:

`-i` Mostra només els datagrames que entren per la interfície a escoltar.

`-o` Mostra només els datagrames que surten per la interfície a escoltar.

`-d` Rep, com a argument, el nom de la interfície a escoltar. Si no se n'indica cap, llavors s'escolten totes les interfícies de xarxa.

`-s` Mostra alguna informació de la capçalera SCTP.

`-t` Mostra alguna informació de la capçalera TCP.

`-u` Mostra alguna informació de la capçalera UDP.

Per defecte, el programa mostrarà únicament les adreces IP origen i destí dels datagrames. En cas d'estar actives les opcions *s*, *t* o *u*, llavors es mostraran també els números de port.

Convé recordar que aquest programa s'ha d'executar amb permisos de superusuari.

```
1  // sockets/sniffer/sniffer.c
2  #define _GNU_SOURCE
3  #include <arpa/inet.h>
4  #include <net/if.h>
5  #include <netinet/if_ether.h>
6  #include <netpacket/packet.h>
7  #include <sys/ioctl.h>
8  #include <sys/socket.h>
9  #include <getopt.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include <strings.h>
14 #include <unistd.h>
15 #include "sniffer.h"
16
17 static void parse_arguments (int, char *[]);
18 static int open_socket (void);
19 static void bind_device (const int);
20 static void print_frame (const char *);
```

```
21 static char *device;
22 static int in_option;
23 static int out_option;
24 static int tcp_option;
25 static int udp_option;
26 static int sctp_option;
27 static int mtu = 1500;
28
29 int
30 main (int argc, char *argv[])
31 {
32   parse_arguments (argc, argv);
33
34   int sock_fd = open_socket ();
35   char buffer[mtu];
36   struct sockaddr_ll addr;
37   socklen_t len = sizeof (addr);
38
39   while (1)
40     {
41       recvfrom (sock_fd, buffer, mtu, 0, (void *) &addr, &len);
42
43       if (in_option && addr.sll_pkttype != PACKET_HOST)
44         continue;
45       if (out_option && addr.sll_pkttype != PACKET_OUTGOING)
46         continue;
47
48       print_frame (buffer);
49     }
50 }
51
52 static void
53 parse_arguments (int argc, char *argv[])
54 {
55   int c;
56   while ((c = getopt (argc, argv, "d:iostu")) != -1)
57     {
58       switch (c)
59         {
60         case 'd':
61           if (optarg)
62             device = strdup (optarg);
63           break;
64
65         case 'i':
66           in_option = 1;
67           out_option = 0;
68           break;
69
```

```
70          case 'o':
71            out_option = 1;
72            in_option = 0;
73            break;
74
75          case 's':
76            sctp_option = 1;
77            break;
78
79          case 't':
80            tcp_option = 1;
81            break;
82
83          case 'u':
84            udp_option = 1;
85            break;
86          }
87      }
88  }
89
90  static int
91  open_socket (void)
92  {
93    int sock_fd = socket (AF_PACKET, SOCK_DGRAM, htons (ETH_P_IP));
94    if (sock_fd < 0)
95      {
96        perror ("socket");
97        exit (EXIT_FAILURE);
98      }
99
100   if (device)
101     bind_device (sock_fd);
102
103   return sock_fd;
104 }
105
106 static void
107 bind_device (const int sock_fd)
108 {
109   struct ifreq ifr;
110
111   strncpy (ifr.ifr_name, device, IFNAMSIZ);
112   if (ioctl (sock_fd, SIOCGIFINDEX, &ifr) < 0)
113     {
114       perror ("ioctl");
115       exit (EXIT_FAILURE);
116     }
117
118   struct sockaddr_ll addr;
```



```
119   bzero (&addr, sizeof (addr));
120
121   addr.sll_family = AF_PACKET;
122   addr.sll_protocol = htons (ETH_P_IP);
123   addr.sll_ifindex = ifr.ifr_ifindex;
124
125   if (bind (sock_fd, (void *) &addr, sizeof (addr)) < 0)
126     {
127       perror ("bind");
128       exit (EXIT_FAILURE);
129     }
130
131   if (ioctl (sock_fd, SIOCGIFMTU, &ifr) > -1)
132     mtu = ifr.ifr_mtu;
133 }
134
135 static void
136 print_frame (const char *buffer)
137 {
138   char *str;
139   char *ip_source = strdup (inet_ntoa (*IP_SOURCE (buffer)));
140   char *ip_dest = strdup (inet_ntoa (*IP_DEST (buffer)));
141
142   if (tcp_option && IP_PROTOCOL (buffer) == IPPROTO_TCP)
143     {
144       uint16_t sport = ntohs (*(buffer + IP_HEADER_LENGTH (buffer) * 4));
145       uint16_t dport = ntohs (*(buffer + IP_HEADER_LENGTH (buffer) * 4 + 2));
146       asprintf (&str, "[%s]:[%hu] --> [%s]:[%hu] IP|TCP (%hu bytes)",
147                 ip_source, sport, ip_dest, dport, IP_TOTAL_LENGTH (buffer));
148     }
149   else if (sctp_option && IP_PROTOCOL (buffer) == IPPROTO_SCTP)
150     {
151       uint16_t sport = ntohs (*(buffer + IP_HEADER_LENGTH (buffer) * 4));
152       uint16_t dport = ntohs (*(buffer + IP_HEADER_LENGTH (buffer) * 4 + 2));
153       asprintf (&str, "[%s]:[%hu] --> [%s]:[%hu] IP|SCTP (%hu bytes)",
154                 ip_source, sport, ip_dest, dport, IP_TOTAL_LENGTH (buffer));
155     }
156   else if (udp_option && IP_PROTOCOL (buffer) == IPPROTO_UDP)
157     {
158       uint16_t sport = ntohs (*(buffer + IP_HEADER_LENGTH (buffer) * 4));
159       uint16_t dport = ntohs (*(buffer + IP_HEADER_LENGTH (buffer) * 4 + 2));
160       asprintf (&str, "[%s]:[%hu] --> [%s]:[%hu] IP|UDP (%hu bytes)",
161                 ip_source, sport, ip_dest, dport, IP_TOTAL_LENGTH (buffer));
162     }
163   else
164     asprintf (&str, "[%s] --> [%s] IP (%hu bytes)", ip_source, ip_dest,
165               IP_TOTAL_LENGTH (buffer));
166
167   printf ("%s\n", str);
```

```
168   free (str);
169   free (ip_dest);
170   free (ip_source);
171 }
```

El codi font de `sniffer.h` es mostra a continuació:

```
1  // sockets/sniffer/sniffer.h
2  #ifndef __SNIFFER_H__
3  #define __SNIFFER_H__
4
5  #define IP_VERSION(a) (((a)[0] & 0xf0) >> 4)
6  #define IP_HEADER_LENGTH(a) (unsigned short)(((a)[0] & 0x0f) * 4)
7  #define IP_TYPE_OF_SERVICE(a) (buffer[1])
8  #define IP_TOTAL_LENGTH(a) (ntohs (*(short *)((a) + 2)))
9  #define IP_IDENTIFICATION(a) (ntohs (*(short *)((a) + 4)))
10 #define IP_FLAGS(a) ((a)[6] & 0xe)
11 #define IP_FRAG_OFFSET(a) (ntohs (*((short *)((a) + 6)) & 0x1fff))
12 #define IP_MAX_HOP(a) ((a)[8])
13 #define IP_PROTOCOL(a) ((a)[9])
14 #define IP_HEADER_CSK(a) (ntohs (*(short *)((a) + 10)))
15 #define IP_SOURCE(a) (struct in_addr *)((a) + 12)
16 #define IP_DEST(a) (struct in_addr *)((a) + 16)
17 #define IP_OPTS(a) (*((int *)((a) + 20)) & 0xfff0)
18 #define IP_PADDING(a) ((a)[23])
19
20 #endif
```

13.3 Operacions sobre interfícies

A part de les funcions relacionades amb sockets i opcions de sockets, en UNIX existeix una funció, anomenada `ioctl`, que permet realitzar una gran varietat de funcionalitats. En aquest apartat s'estudiaran algunes d'aquestes funcionalitats, directament relacionades amb les interfícies de xarxa, com ara obtenir el seu MTU, la màscara de xarxa o l'adreça MAC, per posar alguns exemples.

La funció `ioctl` rep els següents arguments:

- Un file descriptor de tipus socket, és a dir, obert amb la funció `socket`. No acostuma a importar quin protocol s'ha especificat per al socket, generalment és suficient que siguin de la família `AF_INET`.
- Un enter especificant l'operació que es vol realitzar. Hi ha tota una sèrie de `#defines` indicant noms d'operacions, que s'aniran veient al llarg d'aquest apartat.
- Un punter a un objecte, el tipus del qual dependrà de l'operació que es vulgui realitzar. Per a la majoria d'operacions que es veuran en aquest apartat, es tractarà de punters a objectes de tipus `struct ifreq`.

No és l'objectiu d'aquest apartat descriure completament totes les funcionalitats que proveeix `ioctl`, sinó algunes que s'han considerat interessants. L'alumne interessat en el tema pot consultar el capítol 17 de Stevens et al. [2004] i les pàgines manual de `socket(7)`, `tcp(7)`, `packet(7)` i sobretot `netdevice(7)` per a més informació.

13.3.1 Obtenció de l'MTU

L'exemple següent mostra com emprar la funció `ioctl` per a obtenir, des d'un programa en C, el paràmetre MTU d'una interfície (la funció rep el nom de la interfície, com ara `"eth0"`).

```
1  // sockets/get_mtu.c
2  #include <net/if.h>
3  #include <sys/socket.h>
4  #include <sys/ioctl.h>
5  #include <stdio.h>
6  #include <string.h>
7  #include <unistd.h>
8
9  int
10 get_mtu (const char *iface)
11 {
12   int aux = socket (AF_INET, SOCK_STREAM, 0);
13   if (aux < 0)
14     {
15       perror ("get_mtu: socket");
16       return -1;
17     }
18
19   struct ifreq ifr;
20   strncpy (ifr.ifr_name, iface, IFNAMSIZ);
21   if (ioctl (aux, SIOCGIFMTU, &ifr, sizeof (ifr)) < 0)
22     {
23       perror ("get_mtu: ioctl (SIOCGIFMTU)");
24       return -1;
25     }
26   close (aux);
27
28   return ifr.ifr_mtu;
29 }
```

13.3.2 Obtenció de la màscara de xarxa

L'exemple següent mostra com emprar la funció `ioctl` per a obtenir, des d'un programa en C, la màscara de xarxa d'una interfície. La funció rep el nom d'una interfície i un punter a una estructura `struct in_addr`, que conté un únic camp: un enter de 32 bits. La funció retorna -1 si no aconsegueix obtenir la màscara de

xarxa; retorna 1 si l'aconsegueix obtenir i a la zona de memòria apuntada pel segon paràmetre hi haurà la màscara de xarxa.

```
// sockets/get_netmask.c
#include <net/if.h>
#include <netinet/in.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <assert.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int
get_netmask (const char *device, struct in_addr *netmask)
{
  assert (device);
  assert (netmask);

  struct ifreq ifr;
  int fd = socket (AF_INET, SOCK_STREAM, 0);
  if (fd < 0)
    {
      perror ("get_ip_netmask: socket");
      return -1;
    }

  ifr.ifr_addr.sa_family = AF_INET;
  strncpy (ifr.ifr_name, device, IFNAMSIZ);

  if (ioctl (fd, SIOCGIFNETMASK, &ifr) < 0)
    {
      close (fd);
      perror ("get_ip_netmask: ioctl");
      return -1;
    }

  struct sockaddr_in *saddr = (void *) &ifr.ifr_netmask;
  netmask->s_addr = saddr->sin_addr.s_addr;

  return 1;
}
```

13.3.3 Obtenció de l'adreça MAC

L'exemple següent mostra com emprar la funció `ioctl` per a obtenir, des d'un programa en C, l'adreça MAC d'una interfície. La funció rep el nom de la interfície i un punter a un objecte de tipus `struct ether_addr`. Aquest tipus conté un únic

camp, un array de 6 bytes, que contindrà l'adreça MAC. La funció retorna 1 si pot aconseguir l'adreça MAC; retorna -1 en cas contrari.

```
// sockets/get_mac.c
#include <net/ethernet.h>
#include <net/if.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int
get_mac (const char *iface, struct ether_addr *mac)
{
  int aux = socket (AF_INET, SOCK_STREAM, 0);
  if (aux < 0)
    {
      perror ("get_mtu: socket");
      return -1;
    }

  struct ifreq ifr;
  strncpy (ifr.ifr_name, iface, IFNAMSIZ);
  if (ioctl (aux, SIOCGIFHWADDR, &ifr, sizeof (ifr)) < 0)
    {
      perror ("get_mtu: ioctl (SIOCGIFMTU)");
      return -1;
    }
  close (aux);

  bcopy (ifr.ifr_hwaddr.sa_data, mac->ether_addr_octet, 6);
  return 1;
}
```

PART III

Apèndixs

Apèndix A

Manipulació de caràcters, cadenes i buffers de memòria

A.1 Introducció

La finalitat d'aquest capítol és aprofundir els coneixements del lector en una sèrie de parts del llenguatge C relacionades amb la manipulació de caràcters, cadenes i buffers de memòria, que no se solen explicar en d'altres assignatures i que, malgrat tot, formen una bona part de les pràctiques de Sistemes Operatius. D'aquesta manera s'intenta que l'alumne es trobi el mínim de dificultats a l'hora de treballar amb caràcters, cadenes i buffers de memòria, augmentant així la seva productivitat i les possibilitats de poder entregar les pràctiques a temps i funcionant.

Com el seu nom indica, aquest capítol està organitzat en tres grans apartats:

1. L'apartat A.2 introdueix l'alumne en el concepte de caràcter i en les funcions que permeten manipular-los. La majoria de macros i funcions corresponen a les declarades a `ctype.h`.

2. L'apartat A.3 introdueix l'alumne en el concepte de cadena i en les funcions que permeten mostrar-les i manipular-les. La majoria de macros i funcions corresponen a les declarades a `stdio.h` i `string.h`. És molt convenient llegir detingudament aquest apartat, ja que refresca conceptes oblidats, corregeix errors de concepte i aprofundeix i consolida les eines necessàries per a treballar amb cadenes.

3. L'apartat A.4 introdueix l'alumne en el concepte de buffer de memòria i en les funcions que permeten manipular-los. La majoria de macros i funcions

corresponen a les declarades a `string.h` i `strings.h`. Convé que l'alumne presti atenció a aquest apartat, ja que és molt important diferenciar el que és una cadena i el que és un buffer de memòria, en què s'assemblen i quines diferències hi ha.

Es dóna per suposat que l'alumne té un bon coneixement del *llenguatge* C. En cas de no ser així, es recomana que consulti obres com ara Kernighan and Ritchie [1988], Prata [2004] i Harbison and Steele [2002].

A.2 Manipulació de caràcters

A.2.1 Introducció

Un caràcter és un enter de 8 bits. El seu valor numèric no interessa, sinó tan sols el símbol que representa, segons la taula ASCII. En C, una variable de tipus `char` es pot fer servir tant com a caràcter com per a guardar enters de 8 bits. En aquest segon cas, pot ser interessant declarar la variable com a `signed` o `unsigned`, segons si s'admetran nombres positius o no[1].

Moltes de les funcions que treballen amb caràcters accepten arguments i retornen valors de tipus `int`. No hi ha cap mena de problema, ja que en C la conversió entre `char` i `int` és automàtica. Cal tenir en compte, però, d'haver declarat totes les funcions que es cridin.

L'arxiu `ctype.h` declara una sèrie de funcions que poden ser de gran utilitat a l'hora de treballar amb caràcters. Aquestes macros i funcions es poden classificar en dues categories diferents: *classificació* i *conversió*.

A.2.2 Classificació de caràcters

`isalnum` Aquesta funció rep un caràcter, i retorna cert si és un caràcter alfanumèric, és a dir: o és una lletra (tant majúscula com minúscula) o és un número. Aquesta funció és equivalent a

```
isalpha(c) || isdigit (c)
```

`isalpha` Aquesta funció rep un caràcter, i retorna cert si és una lletra, ja sigui majúscula o minúscula. Aquesta funció és equivalent a

```
isupper(c) || islower(c)
```

`isdigit` Aquesta funció rep un caràcter, i retorna cert si és un dígit decimal (és a dir, un dígit en el rang 0–9).

[1] Un enter sense signe admet la mateixa quantitat de valors que la seva versió *signed*; els valors negatius que no hi són, s'afegeixen als positius. Per exemple, un enter amb signe de 16 bits admet valors en el rang $[-32768, 32767]$, la versió senge signe admet valors en el rang $[0, 65535]$.

`isxdigit` Aquesta funció rep un caràcter, i retorna cert si és un dígit hexadecimal, és a dir, un dígit decimal o bé una lletra en el rang `a–f`, tant en majúscules com en minúscules.

`isupper` Aquesta funció rep un caràcter, i retorna cert si és una lletra majúscula.

`islower` Aquesta funció rep un caràcter, i retorna cert si és una lletra minúscula.

`isprint` Aquesta funció rep un caràcter, i retorna cert si és una caràcter que no és un caràcter de control. És a dir, retorna cert per a tots aquells caràcters el codi ASCII dels quals està en el rang 32–126, ambdós inclosos.

`isgraph` Aquesta funció rep un caràcter, i retorna cert per als mateixos casos que la funció `isprint` *excepte* per a l'espai en blanc. És a dir, per a l'espai en blanc `isprint` retorna cert però `isgraph` retorna fals.

`ispunct` Aquesta funció rep un caràcter, i retorna cert si és un signe de puntuació, és a dir: un caràcter per al qual `isprint` retorna cert però no és un espai ni cap altre caràcter per al qual `isalnum` retornaria cert.

`isspace` Aquesta funció rep un caràcter, i retorna cert si és un espai, és a dir:

- Un tabulador (`'\t'`)
- Un retorn de carro (`'\r'`)
- Una nova línia (`'\n'`)
- Un tabulador vertical (`'\v'`)
- Un caràcter de nova pàgina (`'\f'`)
- Un espai en blanc (`' '`)

`isblank` Aquesta funció rep un caràcter, i retorna cert si és un espai en blanc (`' '`) o un tabulador (`'\t'`).

`isascii` Aquesta funció rep un caràcter, i retorna cert per a tots aquells caràcters el codi ASCII dels quals està en el rang 0–127, ambdós inclosos.

`iscntrl` Aquesta funció rep un caràcter, i retorna cert per a tots aquells caràcters el codi ASCII dels quals està en el rang 0–31, ambdós inclosos.

A.2.3 Conversió de caràcters

`tolower` Per a un caràcter en majúscula, aquesta funció retorna el seu corresponent caràcter en minúscula.

`toupper` Per a un caràcter en minúscula, aquesta funció retorna el seu corresponent caràcter en majúscula.

`toascii` Aquesta funció rep un caràcter i retorna un altre caràcter:

- Si el caràcter rebut està comprès en el rang 0–127, el caràcter retornat és igual al caràcter rebut.
- En cas contrari, el caràcter retornat és el caràcter rebut però amb el bit de més pes posat a zero.

`toint` Per als caràcters pels quals `isxdigit` retornaria cert, aquesta funció retorna:

- Per als caràcters compresos entre `'0'` i `'9'`, retorna 0–9 respectivament.
- Per als caràcters compresos entre `'a'` i `'f'` (tant en majúscules com en minúscules), retorna 10–15 respectivament.

Per a la resta de caràcters, el valor de retorn és *implementation-defined*.

A.3 Manipulació de cadenes

A.3.1 Introducció

Una cadena és una seqüència de bytes imprimibles, que finalitza amb un byte el valor del qual és 0 (típicament representat com a `'\0'`). Per caràcters imprimibles s'entén caràcters que es poden mostrar per pantalla, o sigui per als quals la funció `isprint` retornaria cert.

Totes aquelles funcions que diguin que treballin amb cadenes esperaran que hi hagi el caràcter de final de cadena. Si no el troben, es poden produir resultats molt desagradables. En el millor dels casos es produiran *segmentation faults*; en el pitjor, es poden sobreescriure altres variables o regions de memòria, que provocaran errors que seran enormement difícils de *debuggar*.

Des del punt de vista del llenguatge C, una cadena de caràcters pot ser un array de caràcters o un punter a caràcter:

```
char str1[] = "This is a string";
char *str2 = "This is another string";
```

Cal tenir en compte que la cadena `str1` es pot modificar, mentre que la cadena `str2` no:

- La variable `str1` és un array de caràcters. El signe d'igualtat *no és* una assignació, sinó que és una inicialització: concretament inicialitza tots i cadascun dels elements de l'array, incloent espai per al caràcter finalitzador de cadena.

- La variable `str2` és un punter a caràcter. El signe d'igualtat *no és* una assignació, sinó que és una inicialització: indica que la variable `str2` apuntarà a una zona de memòria que conté un literal de cadena, que en C és un *non-modifiable left-value*, és a dir, que `str2` apunta a una zona de memòria que *no es pot modificar*.

Dins de les funcions que operen amb cadenes, distingirem entre aquelles que no alteren els seus arguments i aquelles que sí els alteren.

A.3.2 Duplicat de cadenes

Hi ha dues funcions per a duplicar cadenes:

```
#include <string.h>

char *strdup (const char *s);
char *strndup (const char *s, size_t n);
```

- La funció `strdup` rep un punter a cadena, i retorna un punter a una altra cadena, idèntica a l'original. Modificar-ne una no suposa modificar l'altra. El punter retornat es pot alliberar amb `free`.

- La funció `strndup` realitza la mateixa acció que `strdup`, però copia com a molt `n` caràcters.

Les dues funcions s'encarregen de posar el caràcter de finalització de cadena (`'\0'`) a la cadena que retornen.

A.3.3 Tractament de cadenes *read-only*

`strlen` La funció `strlen` retorna quants caràcters conté la cadena *sense* incloure el caràcter finalitzador. La seva declaració és la següent:

```
#include <string.h>

size_t strlen (const char *);
```

Comparació de cadenes Per a comparar cadenes es disposa de les següents funcions:

```
#include <string.h>

int strcmp (const char *s1,
            const char *s2);
int strncmp (const char *s1,
             const char *s2,
             size_t len);

#include <strings.h>

int strcasecmp (const char *s1,
                const char *s2);
int strncasecmp (const char *s1,
                 const char *s2,
                 size_t len);
```

- La funció `strcmp` compara, caràcter a caràcter, les dues cadenes. Quan es troba una diferència entre elles, la comparació finalitza. La comparació també finalitza, evidentment, quan s'arriba al final d'una de les dues cadenes.

  La funció `strcmp` retorna zero si les dues cadenes són idèntiques; retorna un altre valor diferent de zero en cas que siguin diferents.

- La funció `strncmp` funciona exactament igual que `strcmp`, però compara, com a *màxim*, `len` caràcters.

- La funció `strcasecmp` funciona exactament igual que `strcmp`, però al comparar no distingeix entre majúscules i minúscules.

- La funció `strncasecmp` funciona exactament igual que `strcasecmp`, però compara, com a *màxim*, `len` caràcters.

`strchr` i `strrchr` Les dues funcions estan declarades de la següent manera:

```
#include <string.h>

char *strchr (const char *s, int c);
char *strrchr (const char *s, int c);
```

- La funció `strchr` busca el caràcter `c` a la cadena `s` (la cerca inclou també el caràcter finalitzador de cadena), i retorna un punter a la primera ocurrència de tal caràcter.

- La funció `strrchr` busca el caràcter `c` a la cadena `s` (la cerca inclou també el caràcter finalitzador de cadena), i retorna un punter a l'última ocurrència de tal caràcter, és a dir, comença la cerca pel final.

Si qualsevol d'aquestes dues funcions no troben el caràcter que estan buscant, retornen un punter `NULL`. A continuació s'inclou, com a exemple, una funció que calcula quantes vegades hi ha, en una cadena, un determinat caràcter:

```
1  int
2  how_many (const char *s, int c)
3  {
4    int n = 0;
5    if (c == 0)
6      return 0;
7    while (s != NULL)
8      {
9        s = strchr (s, c);
10       if (s != NULL)
11         n++, s++;
12     }
13   return n;
14 }
```

`strstr` La funció `strstr` està declarada de la següent manera:

```
#include <string.h>

char *strstr (const char *src, const char *sub);
```

La finalitat d'aquesta funció és buscar, dins de la cadena `src`, si hi ha la cadena `sub`. En cas afirmatiu, retorna un punter al principi de tal ocurrència. En cas negatiu, retorna un punter `NULL`.

Conversió de cadena a nombre Per a convertir el valor numèric d'una cadena a una variable de tipus aritmètic, es disposa de les funcions següents:

```
#include <stdlib.h>

double atof (const char *str);
int atoi (const char *str);
long int atol (const char *str);
long long int atoll (const char *str);
```

L'única diferència entre aquestes funcions és en si accepten decimals o no (`atof` els accepta, la resta no) i en la mida de l'enter que retornen (per a `atoi`, `atol` i `atoll`).

A.3.4 Manipulació de cadenes

Còpia i concatenació de cadenes Les dues funcions "clàssiques" per a comparar i concatenar cadenes són `strcpy` i `strcat`, respectivament:

```
#include <string.h>

char *strcat (char *dest, const char *src);
char *strcpy (char *dest, const char *src);
```

- La funció `strcpy` copia el contingut de la cadena `src` *a sobre* del contingut de la cadena `dest`. Copia tants caràcters com trobi a la cadena `src`, incloent el caràcter de finalització de cadena. El valor de retorn de la funció és `dest`.
- La funció `strcat` copia el contingut de la cadena `src` *al final* del contingut de la cadena `dest`, sobreescrivint el caràcter de final de cadena. Copia tants caràcters com trobi a la cadena `src`, incloent el caràcter de finalització de cadena. El valor de retorn de la funció és `dest`.

Aquestes dues funcions tenen un problema: no comproven que la cadena de destí sigui suficientment gran per encabir-hi el resultat de l'operació. Per exemple:

```
char origen[200];
char desti[20];

scanf("\n%s", origen);
strpcy(desti, origen);
```

Si la cadena `origen` conté més de 20 caràcters, la cadena `desti` no podrà encabir-hi el resultat. La funció `strcat` presenta un problema similar. Per tal de solucionar aquest problema, es disposa de les dues funcions següents:

```
#include <string.h>

char *strncpy(const char *dst, const char *src,
              size_t len);
char *strncat(const char *str, const char *append,
              size_t count);
```

L'últim paràmetre d'aquestes dues funcions indica:

- Per a la funció `strncpy`, el número màxim de caràcters de `src` que es copiaran a `dst`.
- Per a la funció `strncat`, el número màxim de caràcters de `append` que s'afegiran al final de `str`.

Aquestes funcions de còpia i concatenació, encara que retornen un valor, és millor prescindir d'aquest valor de retorn, gairebé sempre inútil.

A.3.5 Escriptura a cadenes

Pot interessar, en determinades circumstàncies, escriure en una cadena de la mateixa manera que ho fa la funció `printf`. A tal fi, es disposa de les següents funcions:

```
#include <stdio.h>

int printf (const char *format, ...);
int sprintf (char *str, const char *format, ...);
int snprintf (char *str, size_t size, const char *format, ...);
int asprintf (char **str, const char *format, ...);
```

- La funció `sprintf` té un comportament idèntic al de la funció `printf`, però el que s'escriuria per pantalla s'escriu a la cadena apuntada per `str`. No es comprova que aquesta cadena tingui espai suficient per a encabir-hi tota la sortida. Les funcions `printf` i `sprintf` retornen la quantitat de caràcters escrits (ja sigui a pantalla o a una cadena), sense incloure el caràcter finalitzador de cadena.

- La funció `snprintf` té un comportament idèntic al de la funció `sprintf`, però no escriu més de `size` caràcters a la cadena apuntada per `str`. Aquesta funció retorna el número de caràcters que *s'haurien* escrit si la cadena `str` tingués espai suficient per a encabre'ls. És a dir, si el valor de retorn de la funció és superior a `size`, llavors la funció `snprintf` no ha pogut escriure, a la cadena `str`, tot el que hi havia d'escriure.

- La funció `asprintf` té un comportament idèntic al de la funció `sprintf`, però és la pròpia funció la que s'encarrega de demanar memòria per a la cadena, de manera que s'evita el problema de fer un *overflow*. Aquesta memòria la demana cridant la funció `malloc`, per tant el programador és responsable d'alliberar-la. La funció retorna la quantitat de caràcters escrits, sense incloure el caràcter finalitzador de cadenes.

  Si, per algun motiu, `asprintf` no pot aconseguir memòria, retorna -1 i el contingut del punter `str` és indefinit (FreeBSD l'estableix a `NULL`).

Avís Alguns programes fan servir la funció `sprintf` de la següent manera:

```
sprintf (buf, "%s some more chars", buf);
```

per tal d'afegir caràcters a la cadena `buf`. L'estàndar de C diu, explícitament, que si la cadena origen i destí se solapen, els resultats són indefinits, la qual cosa significa que els resultats obtinguts poden no ser els que s'esperaven.

A.3.6 Parsejat de cadenes

Alguns anys s'ha demanat a la pràctica fer un client que implementés certes funcionalitats d'un *shell*, és a dir, que permetés executar comandes de UNIX d'una forma similar (però amb moltes menys funcionalitats) que un shell real (com ara `bash`). Típicament s'ha demanat executar comandes amb un número arbitrari d'arguments i, com a màxim, un nivell de *pipe*. En aquesta mena de problemes se sol partir d'una cadena de caràcters d'una mida prefixada, la qual s'ha d'analitzar convenientment per tal d'executar-la correctament.

No es tracta, en aquesta mena de problemes, de dissenyar una gramàtica incontextual i un parser ascendent o descendent, d'una manera similar a com es faria en la construcció d'un compilador. Tenint en compte el que se sol demanar a les pràctiques de Sistemes Operatius, el parser es pot construir d'una manera molt més simple i sense haver de recórrer a conceptes propis d'altres assignatures.

La funció que aquí es presenta, `strtok`, ha demostrat ser extremadament útil per a aquest propòsit. La seva declaració és la següent:

```
#include <string.h>

char *strtok(char *str, const char *sep);
```

El significat de cada paràmetre és el següent:

`str` Punter a la cadena de caràcters que es vol parsejar. A la primera crida, es passa el punter a la cadena. En les següents crides (en què s'hagi de parsejar la mateixa cadena), es passa `NULL`.

`sep` Cadena on cadascun dels caràcters són els separadors.

El funcionament de la funció és força senzill:

1. La funció `strtok` se salta els caràcters separadors (indicats a l'argument `sep`) fins a trobar un caràcter que no és separador.
2. Quan troba un caràcter que no és separador, segueix avançant en la cadena i subsitueix el primer separador que troba per `'\0'`.
3. Immediatament després retorna un punter al caràcter no separador que ha trobat.
4. Quan ja no hi ha més caràcters que no siguin separadors, retorna `NULL`.

Per exemple, si els separadors són els espais en blanc i la cadena és "pràctiques de sistemes operatius", cridarem la funció `strtok` diverses vegades:

1. La primera crida retornarà un punter a "pràctiques" i, després de la lletra "s", hi posarà `'\0'` (al lloc de l'espai en blanc).

2. La segona crida retornarà un punter a “de” i, després de la lletra “e”, hi posarà `'\0'` (al lloc de l’espai en blanc).

3. La tercera crida retornarà un punter a “sistemes” i, després de la lletra “s”, hi posarà `'\0'` (al lloc de l’espai en blanc).

4. La quarta crida retornarà un punter a “operatius” i, després de la lletra “s”, hi posarà `'\0'` (al lloc de l’espai en blanc).

5. La cinquena crida retornarà `NULL`.

Com a exemple, es mostra un programa que demana una frase a l’usuari i la separa en paraules:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX 80

int
main (void)
{
  char str[MAX];
  char **ptr, *aux;
  int c;

  printf ("Introdueix una frase:\n");
  fgets (str, MAX, stdin);
  str[strlen (str) - 1] = '\0';

  ptr = malloc (sizeof (char *));
  assert (ptr != NULL);

  for (c = 0, aux = strtok (str, " ");
       aux != NULL; aux = strtok (NULL, " "), c++)
    {
      ptr = realloc (ptr, sizeof (char *) * (c + 2));
      ptr[c] = aux;
    }
  ptr[c] = NULL;

  for (c = 0, aux = ptr[c]; aux != NULL; c++, aux = ptr[c])
    {
      printf ("Paraula %d: \"%s\"\n", c, aux);
    }

  free (ptr);
```

```
  return EXIT_SUCCESS;
}
```

La funció `strtok` té els següents *issues*:

La funció `strtok` té un problema: té memòria interna, i pot portar problemes si es volen parsejar, simultàniament, diferents cadenes. A més a més, no és una funció *thread-safe* – és a dir, si una aplicació fa servir threads i dos o més threads criden `strok`, es poden interferir entre ells. Per a tal fi, hi ha la funció `strtok_r`, que rep un tercer argument de tipus `char **`, que representa la *memòria interna*. El següent exemple il·lustra el funcionament de `strtok_r`:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int
main (void)
{
  char test[80], blah[80];
  char *sep = "\\/:;=-";
  char *word, *phrase, *brkt, *brkb;

  strcpy (test, "This;is.a:test:of=the/string\\tokenizer-function.");

  for (word = strtok_r (test, sep, &brkt);
       word; word = strtok_r (NULL, sep, &brkt))
    {
      strcpy (blah, "blah:blat:blab:blag");

      for (phrase = strtok_r (blah, sep, &brkb);
           phrase; phrase = strtok_r (NULL, sep, &brkb))
        {
          printf ("So far we're at %s:%s\n", word, phrase);
        }
    }
  return EXIT_SUCCESS;
}
```

A.4 Manipulació de buffers

Un buffer de memòria és una sèrie consecutiva de bytes. També es pot veure com una regió contígua de memòria apta per a emmagatzemar-hi dades, del tipus que siguin. Totes les cadenes s'emmagatzemen en buffers, però no tots els buffers guarden cadenes. La diferència fonamental és que, en una cadena, el byte zero (`'\0'`) té un significat especial (final de cadena), mentre que en un buffer de memòria, no en té cap: simplement, és un byte més, com qualsevol altre. Per això totes les funcions

que operen amb buffers necessiten rebre, com a argument, la mida del buffer sobre el qual han d'operar.

`memchr` La funció `memchr` està declarada de la següent manera:

```
#include <string.h>

void *memchr (const void *ptr, int val, size_t len);
```

La funció `memchr` busca, a la regió de memòria apuntada per `ptr` (i la mida de la qual és `len` bytes), la primera ocurrència del caràcter `val`. Si aquest caràcter es troba, `memchr` retorna un punter a la primera ocurrència; en cas contrari, retorna `NULL`.

`memcmp` i `bcmp` Les funcions `memcmp` i `bcmp` estan declarades de la següent manera:

```
#include <string.h>

int memcmp (const void *ptr1, const void *ptr2, size_t len);

#include <strings.h>

int bcmp (const void *ptr1, const void *ptr2, size_t len);
```

La finalitat de les dues funcions és comparar les dues regions de memòria, la mida de les quals és `len` bytes. Les funcions retornen zero si les dues regions són idèntiques, i retornen diferent de zero si són diferents.

Còpia de buffers Per a copiar buffers de memòria es disposa de les següents funcions:

```
#include <string.h>

void *memcpy (void *dest, const void *src, size_t n);
void *memmove (void *dest, const void *src, size_t n);

#include <strings.h>

void bcopy (const void *src, void *dest, size_t n);
```

- La funció `memcpy` copien `n` caràcters de la zona de memòria apuntada per `src` a la zona de memòria apuntada per `dest`, sobreescrivint el contingut anterior. Les dues regions de memòria *no es poden solapar*. La funció retorna un punter a `dest`.

- La funció `memmove` fa exactament el mateix que la funció `memcpy`, però les zones de memòria poden estar solapades. La funció retorna un punter a `dest`.
- La funció `bcopy` fa exactament el mateix que la funció `memmove`, però no retorna res i els arguments origen i destí estan intercanviats.

`memset` i `bzero` Les funcions `memset` i `bzero` estan declarades de la següent manera:

```
#include <string.h>

void *memset (void *ptr, int val, size_t len);

#include <strings.h>

void bzero (void *ptr, size_t len);
```

La funció `bzero` escriu `len` zeros a la zona de memòria apuntada per `ptr`. En canvi, la funció `memset` permet especificar quin és el byte amb què es vol omplir la zona de memòria. La funció `bzero` no retorna res, la funció `memset` retorna `ptr`.

Bibliografia

Samuel P. Harbison and Guy L. Steele. *C: A Reference Manual.* Prentice Hall, 5 edition, 2002. ISBN 0-13-089592-x.

Brian B. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 2 edition, 1988. ISBN 0-13-110362-8.

Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook.* No Starch Press, September 2010. ISBN 1-59327-220-0.

Stephen Prata. *C Primer Plus.* Sams, 5 edition, 2004. ISBN 0-672-32696-5.

Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment.* Addison-Wesley Professional, 2 edition, 2008. ISBN 0-321-52594-9.

W. Richard Stevens. *TCP/IP Illustrated, volume I: The Protocols.* Addison-Wesley Professional, 1994. ISBN 0-201-63346-9.

W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming volume I.* Addison-Wesley Professional, 3 edition, 2004. ISBN 0-13-141155-1.

Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated, volume II: The Implementation.* Addison-Wesley Professional, 1995. ISBN 0-201-63354-X.

Epíleg

> Whatever I have up till now accepted as most true and assured I have gotten either from the senses or through the senses. But from time to time I have found that the senses deceive, and it is prudent never to trust completely those who have deceived us even once.
>
> — René Descartes, *Meditations On First Philosophy*

If there ever were a quote that described programming with C, it would be this. To many programmers, this makes C scary and evil. It is the Devil, Satan, the trickster Loki come to destroy your productivity with his seductive talk of pointers and direct access to the machine. Then, once this computational Lucifer has you hooked, he destroys your world with the evil "segfault" and laughs as he reveals the trickery in your bargain with him.

But, C is not to blame for this state of affairs. No my friends, your computer and the Operating System controlling it are the real tricksters. They conspire to hide their true inner workings from you so that you can never really know what is going on. The C programming language's only failing is giving you access to what is really there, and telling you the cold hard raw truth. C gives you the red pill. C pulls the curtain back to show you the wizard. **C is truth**.

Why use C then if it's so dangerous? Because C gives you power over the false reality of abstraction and liberates you from stupidity.

— `http://c.learncodethehardway.org/book/introduction.html`