Programmation C et Système
Communication Inter-Processus (IPC)

Régis Barbanchon

L2 Informatique

IPC : Inter-Process Communication

Nous allons voir **trois mécanismes de IPC**,
c-à-d, trois moyens de communiquer entre processus :

- les **tubes anonymes** (ou **pipes**), créés avec `pipe()`,
  pour les processus qui on un ancètre commun,
  typiquement, un parent et son enfant après `fork()`.

- les **tubes nommés** (ou **fifos**), créés avec `mkfifo()`,
  et qui ont une entrée dans un répertoire,
  pour les processus qui ne se connaissent pas.

- l'envoi de **signaux** avec `kill()`, `raise()`, `alarm()`
  et le traitement à leur leur reception par un **handler**
  installé préalablement par `signal()` ou `sigaction()`.

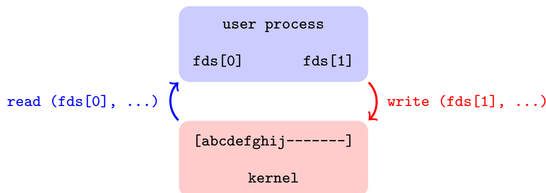
Le tube anonyme : création avec pipe()

Le tube anonyme ou **pipe** est un mécanisme de communication unidirectionnel entre processus qui ont un ancètre commun via fork(). Un pipe est créé via la fonction C éponyme pipe() :

```
#include <unsitd.h>
int pipe (int fds[2]); // returns 0 if OK, -1 on error
```

Deux descripteurs sont ouverts sur le même flot de données (une file de caractères de capacité fixe) qui transite via le kernel, sur lesquels on utilise read(), write(), close().

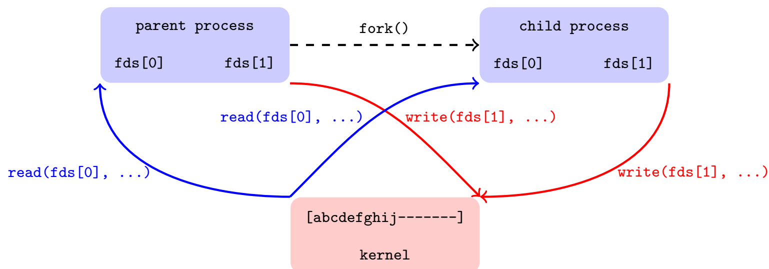
- fds[0] est ouvert en **lecture** sur le pipe,
- fds[1] est ouvert en **écriture** sur le pipe.



Le tube anonyme : pipe() suivi fork()

Un pipe ouvert par un seul processus ne sert à rien.
On utilise fork() pour que le processus parent et enfant partagent le pipe et les fds préalablement ouverts.

Voici la situation lorsqu'un pipe() est suivi d'un fork() :



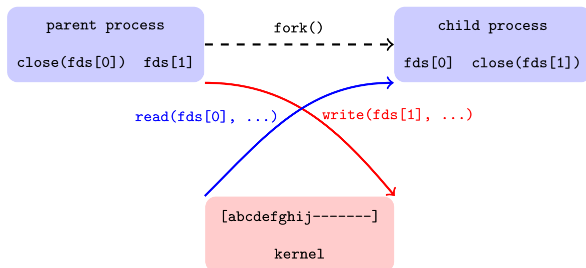
Cependant, l'un des deux processus est toujours l'écrivain
et l'autre processus et le lecteur du pipe,
et chacun doit fermer le descripteur qu'il n'utilise pas.

Le tube anonyme ou pipe : `close()` sur fds inutilisés (1/2)

Dans le cas ou le parent est l'écrivain et l'enfant est le lecteur :

- le parent effectue `close (fds[0])` puisqu'il ne lit pas,
- l'enfant effectue `close (fds[1])` puisqu'il n'écrit pas.

On obtient alors la situation suivante :

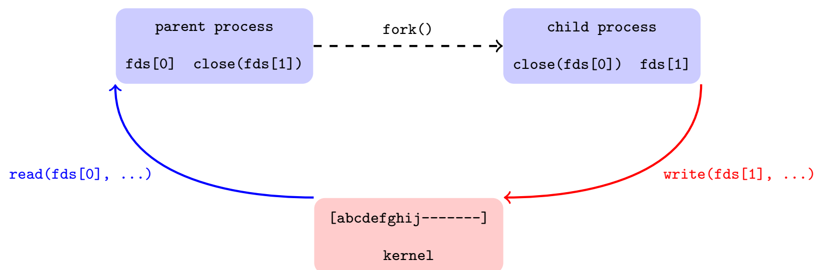


Le tube anonyme ou pipe : `close()` sur fds inutilisés (2/2)

Inversement, si le parent est le lecteur et l'enfant est l'écrivain :

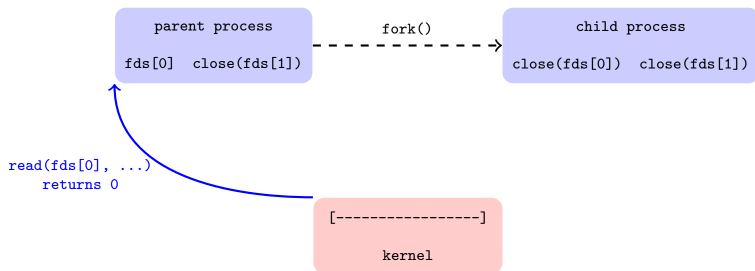
- l'enfant effectue `close (fds[0])` puisqu'il ne lit pas,
- le parent effectue `close (fds[1])` puisqu'il n'écrit pas.

On obtient alors la situation suivante :



Le tube anonyme ou pipe : plus d'écrivain en face

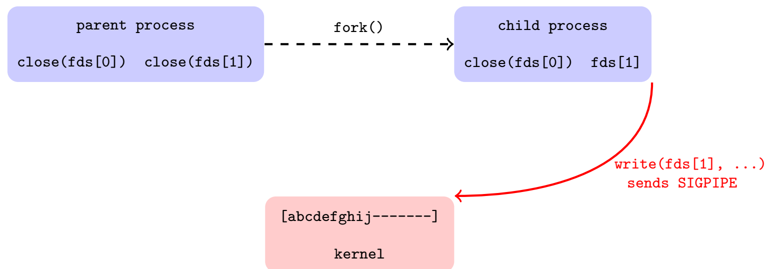
S'il n'y a plus d'écrivain (c-à-d tous les `fds[1]` sont fermés), et qu'il n'y a plus de données dans le pipe, alors `read()` sur `fds[0]` retourne `0` (fin de lecture).



Il est donc important que le lecteur ferme son `fds[1]` inutilisé, car s'il reste ouvert, la fin de la lecture ne pourra être détéctée, un faux écrivain étant toujours à l'autre bout du pipe.

Le tube anonyme ou pipe : plus de lecteur en face

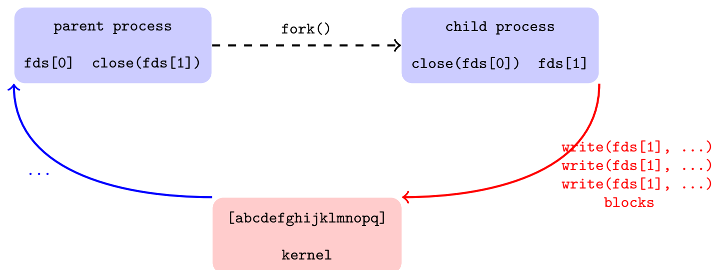
S'il n'y a plus de lecteur (c-à-d tous les `fds[0]` sont fermés), alors un `write()` sur `fds[1]` génère un signal `SIGPIPE` (*broken pipe*).
Le handler par défaut de ce signal termine le processus.
Mais si on le change, avec par ex `signal (SIGPIPE, SIG_IGN);` alors `write()` retourne `-1`, et `errno` vaut `EPIPE`.



Il est donc important que l'écrivain ferme son `fds[0]` inutilisé, car s'il reste ouvert, le *broken pipe* n'est jamais détecté, un faux lecteur étant toujours à l'autre bout du pipe.

Le tube anonyme ou pipe : écriture bloquante

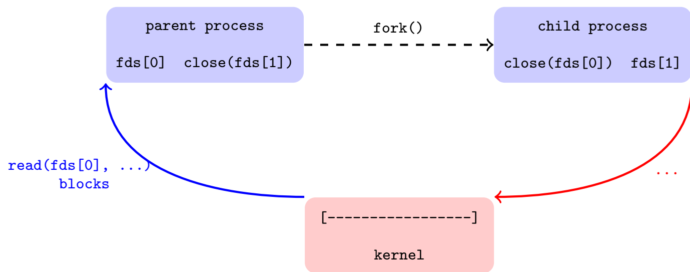
Le pipe étant de capacité finie,
si l'écrivain sature le pipe en écrivant plus vite que le lecteur lit,
il sera bloqué sur un `write()`, en attente d'un `read()` du lecteur.



Cependant, au retour de `write()` bloquant, l'écriture est complète,
et la valeur retournée est `n` si l'on a demandé d'écrire `n` bytes.
Il n'y a d'écriture partielle dans un pipe que si l'on configure `fds[1]` *comme non-bloquant avec la fonction* `fcntl()`.

Le tube anonyme ou pipe : lecture bloquante

S'il n'y a plus de donnée dans le pipe mais qu'il y a un écrivain, le lecteur sera bloqué sur un `read()`, en attente d'un `write()`.



Contrairement à `write()`, le nb de bytes lus au retour de `read()` peut être inférieur à celui-demandé, pas seulement en fin de pipe, mais parce que l'écrivain n'est pas assez rapide pour le lecteur.

Exemple de parent lecteur et d'enfant écrivain (1/5)

Le prog ci-dessous illustre la combinaison `pipe()` + `fork()` :

```
// File: pipefork.c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <assert.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>

int main (void) {
  int fds[2];
  if (pipe (fds) == -1) { perror ("pipe()"); exit (1); }
  pid_t leg= fork();
  if (leg == -1) { perror ("fork()"); exit (1); } // fds closed on exit()
  if (leg ==  0) {
    close (fds[0]);                    // child does not read from pipe
    runChildWriterAndExit (fds[1]); // this function never returns
    assert (false);
  }
  close (fds[1]);                    // parent does not write to pipe
  runParentReaderAndExit (fds[0]); // this function never returns
  assert (false);
  return 0;
}
```

Les fds inutilisés sont fermés puis les I/O sont déléguées à `runParentReaderAndExit()` et `runChildWriterAndExit()`.

Exemple de parent lecteur et d'enfant écrivain (2/5)

Le parent ci-dessous lit les données par blocs d'au plus 3 bytes :

```
void runParentReaderAndExit (int readFd) {
  size_t READ_CAPACITY= 3;
  char buffer [READ_CAPACITY + 1];
  for (;;) {
    ssize_t count= read (readFd, buffer, READ_CAPACITY);
    if (count == 0) break;
    buffer [count]= '\0';
    printf ("parent reads %zd bytes: ....... <%s>\n", count, buffer);
  }
  close (readFd);
  printf ("parent waits child\n");  wait (NULL);
  printf ("parent exits\n");        exit (0);
}
```

L'enfant ci-dessous écrit les données par blocs d'au plus 5 bytes :

```
void runChildWriterAndExit (int writeFd) {
  char msg []= "abcdefghijklmnopqrstuvwxyz";
  size_t length= strlen (msg), start= 0, blockSize= 5;
  while (length != 0) {
    size_t requestSize= (blockSize < length) ? blockSize : length;
    ssize_t count= write (writeFd, msg+start, requestSize);
    printf ("child writes %zd bytes: <%.*s>\n", count, (int)count, msg+start);
    start += count; length -= count;
  }
  close (writeFd);
  printf ("child exits\n");  exit (0);
}
```

Exemple de parent lecteur et d'enfant écrivain (3/5)

Voici deux exemples de sorties du programme :

```
$ ./pipefork
child writes 5 bytes: <abcde>
child writes 5 bytes: <fghij>
parent reads 3 bytes: ....... <abc>
child writes 5 bytes: <klmno>
child writes 5 bytes: <pqrst>
parent reads 3 bytes: ....... <def>
child writes 5 bytes: <uvwxy>
parent reads 3 bytes: ....... <ghi>
child writes 1 bytes: <z>
parent reads 3 bytes: ....... <jkl>
child exits
parent reads 3 bytes: ....... <mno>
parent reads 3 bytes: ....... <pqr>
parent reads 3 bytes: ....... <stu>
parent reads 3 bytes: ....... <vwx>
parent reads 2 bytes: ....... <yz>
parent waits child
parent exits
$
```

```
$ ./pipefork
child writes 5 bytes: <abcde>
parent reads 3 bytes: ....... <abc>
child writes 5 bytes: <fghij>
child writes 5 bytes: <klmno>
parent reads 3 bytes: ....... <def>
child writes 5 bytes: <pqrst>
parent reads 3 bytes: ....... <ghi>
child writes 5 bytes: <uvwxy>
parent reads 3 bytes: ....... <jkl>
child writes 1 bytes: <z>
parent reads 3 bytes: ....... <mno>
parent reads 3 bytes: ....... <pqr>
child exits
parent reads 3 bytes: ....... <stu>
parent reads 3 bytes: ....... <vwx>
parent reads 2 bytes: ....... <yz>
parent waits child
parent exits
$
```

Les entrelacements peuvent changer selon les exécutions.

Exemple de parent lecteur et d'enfant écrivain (4/5)

Le parent modifié ci-dessous stoppe au bout de 2 lectures :

```
void runParentReaderAndExit (int readFd) {
  size_t READ_CAPACITY= 3;
  char buffer [READ_CAPACITY + 1];
  for (int k= 0; k < 2; k++) {
    ssize_t count= read (readFd, buffer, READ_CAPACITY);
    if (count == 0) break;
    buffer [count]= '\0';
    printf ("parent reads %zd bytes: ....... <%s>\n", count, buffer);
  }
  close (readFd);
  printf ("parent waits child\n");  wait (NULL);
  printf ("parent exits\n");        exit (0);
}
```

L'enfant modifé ci-dessous détecte le *broken pipe* :

```
void runChildWriter (int writeFd) {
  signal (SIGPIPE, SIG_IGN);
  char msg []= "abcdefghijklmnopqrstuvwxyz";
  size_t length= strlen (msg), start= 0, blockSize= 5;
  while (length != 0) {
    size_t requestSize= (blockSize < length) ? blockSize : length;
    ssize_t count= write (writeFd, msg+start, requestSize);
    if (count == -1) { printf ("child detects broken pipe\n"); break; }
    printf ("child writes %zd bytes: <%.*s>\n", count, (int)count, msg+start);
    start += count; length -= count;
  }
  close (writeFd);
  printf ("child exits\n");  exit (0);
}
```

Exemple de parent lecteur et d'enfant écrivain (5/5)

Voici deux exemples de sorties du programme ainsi modifié :

```
$ ./brokenpipe
child writes 5 bytes: <abcde>
child writes 5 bytes: <fghij>
parent reads 3 bytes: ....... <abc>
child writes 5 bytes: <klmno>
parent reads 3 bytes: ....... <def>
child writes 5 bytes: <pqrst>
parent waits child
child detects broken pipe
child exits
parent exits
$
```

```
$ ./brokenpipe
child writes 5 bytes: <abcde>
child writes 5 bytes: <fghij>
child writes 5 bytes: <klmno>
child writes 5 bytes: <pqrst>
child writes 5 bytes: <uvwxy>
child writes 1 bytes: <z>
parent reads 3 bytes: ....... <abc>
child exits
parent reads 3 bytes: ....... <def>
parent waits child
parent exits
$
```

À gauche, l'enfant détecte un *broken pipe* lorqu'il tente `write (writeFd, "uvwxy", 5)` qui retourne `-1`, à cause de `SIGPIPE` qui est ignoré via `signal (SIGPIPE, SIG_IGN)`.
Le parent bloque ici sur `wait()` jusqu'à la terminaison de l'enfant.

À droite, l'enfant à eu le temps de faire toutes ses écritures avant que le parent fasse ses deux lectures : pas de *broken pipe*.

Duplication de descripteur avec dup2() (1/3)

Lorsque l'enfant effectue un recouvrement par execve(),
on veut en général l'une des deux redirections suivantes :

- que sa sortie standard écrive dans le fds[1] de notre pipe.
- ou que son entrée standard lise le fds[0] de notre pipe.

Il faut rediriger le flux standard désiré. Ceci se fait avec dup2() :

```
#include <unistd.h>
int dup2(int oldFd, int newFd); // returns newfd if OK, -1 on error.
```

La fonction dup2() ferme d'abord le descripteur newFd
puis réutilise le même entier pour en faire une copie de oldFd.
Ainsi newFd et oldFd pointent vers le même flux ouvert.

Pour rediriger stdin (de fd 0) ou stdout (de fd 1) :

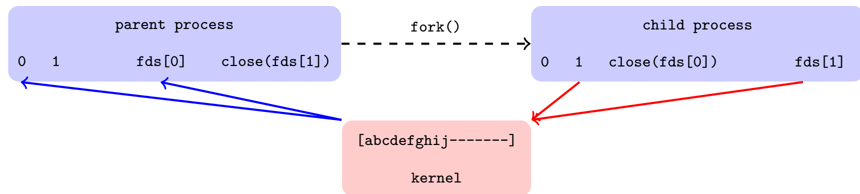
```
dup2 (fds[0], 0);
close(fds[0]); // fds[0] is now useless
```

```
dup2 (fds[1], 1);
close(fds[1]); // fds[1] is now useless
```

Là encore, il faut penser à fermer le descripteur devenu inutile.

Duplication de descripteur avec `dup2()` (2/3)

Si le parent fait `dup2 (fds[0], 0);`
et l'enfant fait `dup2 (fds[1], 1);` alors on a :

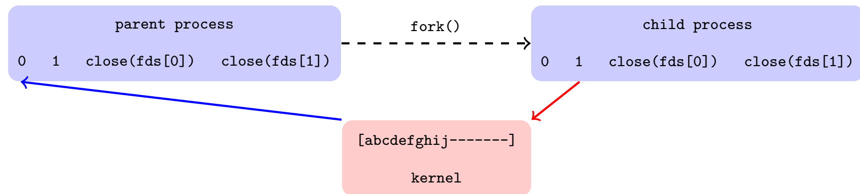


On aura un problème de detection de fin de lecture pour `read()`
et un problème de de détection de *broken pipe* pour `write()`,
à cause des descripteurs multiples ouverts sur les bouts du tube.

Remarque : La fin de lecture sera détectée à la terminaison de l'écrivain, puisque celle-ci ferme tous les descripteurs ouverts.

Duplication de descripteur avec dup2() (3/3)

Si le parent fait `dup2(fds[0], 0); close(fds[0]);`
et l'enfant fait `dup2(fds[1], 1); close(fds[1]);` alors on a :



Le pipe n'a plus qu'un lecteur, l'entrée standard du parent,
et il n'a plus qu'un écrivain, la sortie standard de l'enfant.

Les fins de tube sont donc maintenant détectables :
L'enfant reçoit `SIGPIPE` s'il écrit après que le parent a fermé `fd 0`.
Dans le parent, `read()` retourne `0` si l'enfant a fermé `fd 1`.

Exemple de combinaison pipe()+fork()+execlp()

Le programme suivant est un analogue du pipe `ls *.c | wc -l` :

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

int main (void) {

  int fds[2];
  if (pipe (fds) == -1) { perror ("pipe()"); exit (1); }
  pid_t leg= fork();
  if (leg == -1) { perror ("fork()"); exit (1); } // fds closed on exit()
  if (leg ==  0) {
    close (fds[0]);
    dup2  (fds[1], 1);
    close (fds[1]);
    execlp ("sh", "sh", "-c", "ls *.c", (char *) NULL);
    exit(1); // reached only if execlp() failed.
  }
  close (fds[1]);
  dup2  (fds[0], 0);
  close (fds[0]);
  execlp ("wc", "wc", "-l", (char *) NULL);
  return 1; // reached only if execlp() failed.
}
```

Le parent est le lecteur du pipe. Il exécute `wc -l`.
L'enfant est l'écrivain du pipe. Il exécute `sh -c 'ls *.c'`.

Les fonctions popen() et pclose()

```
#define _XOPEN_SOURCE
#include <stdio.h>
FILE * popen(char const cmd[], char const type[]);
int pclose(FILE * stream);
```

popen() exécute une commande cmd via le shell (sh -c cmd) et donne accès à un de ses flux standard via un FILE* selon le type :

- type "w" : écrit sur l'entrée standard de cmd.
- type "r" : lit la sortie standard de cmd.

En reprenant l'exemple précédent ls *.c | wc -l :

```
#define _XOPEN_SOURCE
#include <stdio.h>

int main (void) {
  FILE * stream= popen ("ls *.c | wc -l",    "r"   );
  int count;
  fscanf (stream, "%d", & count);
  pclose (stream);
  fprintf (stdout, "there are %d source files\n", count);
  return 0;
}
```

Le FILE* doit être fermé avec pclose() au lieu de fclose().

Le tube nommé ou fifo : création avec mkfifo()

Un **tube nommé** ou **fifo** (*First-In First-Out*) est un mécanisme de communication unidirectionnel entre processus qui a une entrée dans un répertoire comme un fichier classique.
On crée un fifo soit avec la commande mkfifo :

```
$ mkfifo -m 0666 somefifo      # 0666 is octal for mode u=rw,g=rw,o=rw
$ ls -l somefifo
prw-rw-rw- 1 regis prof 0 avril  5 14:54 somefifo
$ rm fifoname
```

... soit avec la fonction C éponyme mkfifo() :

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

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

```
// mkfifo.c
int main (void) {
  umask (0);
  if (mkfifo ("somefifo", 0600) == -1) { perror ("mkfifo"); exit (1); }
  fprintf (stderr, "created fifo 'somefifo'\n");
  return 0;
}
```

```
$ ./mkfifo
created fifo 'somefifo'
$ ls -l somefifo
prw------- 1 regis prof 0 avril  5 14:54 somefifo
```

```
$ ./mkfifo
mkfifo: File exists
$ rm somefifo
```

Le tube nommé ou fifo : ouverture avec open()

Un fifo s'ouvre avec `open()` comme un fichier régulier.
Par défaut, l'ouverture est bloquante en lecture et en écriture :

- `int fd= open ("somefifo", O_RDONLY);`
  l'ouverture en lecture **bloque** jusqu'à ce qu'un autre processus ouvre le même fifo en écriture.
- `int fd= open ("somefifo", O_WRONLY);`
  l'ouverture en écriture **bloque** jusqu'à ce qu'un autre processus ouvre le même fifo en lecture.

L'écrivain attend donc un lecteur, et le lecteur attend un écrivain.

Le flag `O_NONBLOCK` rend l'ouverture et les IO **non-bloquantes** :

- `int fd= open ("somefifo", O_RDONLY | O_NONBLOCK);`
  l'ouverture en lecture retourne immédiatement même s'il n'y a pas d'écrivain.
- `int fd= open ("somefifo", O_WRONLY | O_NONBLOCK);`
  l'ouverture en écriture retourne aussi immédiatement, mais échoue avec `fd = -1` si le fifo n'a pas de lecteur.

Envoi de signal avec kill(), raise(), et alarm()

kill() est la fonction C éponyme de la commande kill.

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int signum); // returns 0 if OK,  -1 on error.
int raise (int signum);           // returns 0 if OK, < 0 on error.
```

kill() envoie le signal signum au processus pid.
Lorsque signum = 0, aucun signal n'est envoyé,
mais le test d'erreur permet de vérifier l'existence du processus.

Si le processus appelant n'a qu'un seul thread d'exécution, alors
raise (signum) est un wrapper sur kill (getpid(), signum).
La fonction envoie le signal signum au processus appelant.

alarm(n) fait en sorte que SIGALRM sera envoyé dans n secondes.

```
#include <unistd.h>
unsigned int alarm (unsigned int seconds);
```

Une nouvelle alarme remplace l'éventuelle alarme en cours.
Le nombres de secondes restant pour cette dernière est retourné.
alarm(0) annule toute disposition d'alarme en cours.

Liste des principaux signaux et action par défaut

La table suivante est extraite de la commande `man 7 signal` :

```
Signal  Value  Action       Comment
------  -----  -----------  ---------------------------------------------
 HUP       1    Terminate    Hangup detected on controlling terminal
                             (or death of controlling process)
 INT       2    Terminate    Interrupt from keyboard by CTRL-C

 QUIT      3    Dump Core    Quit from keyboard by CTRL-\
 ILL       4    Dump Core    Illegal Instruction
 TRAP      5    Dump Core    Trace/breakpoint trap
 ABRT      6    Dump Core    Abort signal from abort()
 FPE       8    Dump Core    Floating point exception
 SEGV     11    Dump Core    Invalid memory reference

 KILL      9    Terminate    Kill signal
 USR1     10    Terminate    User-defined signal 1
 USR2     12    Terminate    User-defined signal 2
 PIPE     13    Terminate    Broken pipe: write to pipe with no readers
 ALRM     14    Terminate    Timer signal from alarm()
 TERM     15    Terminate    Termination signal
 STKFLT   16    Terminate    Unused, stack fault

 CHLD     17    Ignore       Child stopped or terminated
 CONT     18    Continue     Continue if stopped
 STOP     19    Stop         Stop process
 TSTP     20    Stop         Stop typed at terminal by CTRL-Z
 TTIN     21    Stop         Terminal  input for background process
 TTOU     22    Stop         Terminal output for background process
```

L'action de la majorité des signaux est de terminer le processus.

Réception de signaux avec `signal()`

À l'exception des signaux `SIGKILL` et `SIGSTOP`,
on peut changer l'action d'un signal avec la fonction `signal()`.

```
#include <signal.h>
typedef void (* sighandler_t) (int signum);
sighandler_t signal (int signum, sighandler_t handler);
```

La fonction `handler` est invoquée à la reception du signal.
Elle n'a pas d'autre arg que le numéro du signal reçu `signum`.

Exemple : (malheureusement non portable, *cf.* plus loin)

```
bool ALARMED= false; // global variable

void handleAlarm (int signum) {
   (void) signum; // unused, signum is expected to be SIGALRM here
   ALARMED= true;
}

int main (void) {
  signal (SIGALRM, handleAlarm);
  ALARMED= false;
  alarm (3); // will interrupt with SIGALRM in about 3 seconds
  ...
  return 0;
}
```

Non portabilité de signal() sur deux aspects

signal() n'est portable qu'avec deux handlers spéciaux :

- SIG_IGN, ignorant le signal.
- SIG_DFL, le handler par défaut (*cf.* la table des signaux).

signal() n'est pas portable pour installer une fonction handler, car on ne sait pas selon les différents systèmes :

- si le handler installé est conservé après exécution, ou si l'action SIG_DFL est réinstallée (*one-shot*).
- si un appel système bloquant qui est interrompu est repris, ou s'il est débloqué en retournant -1 avec errno= EINTR.

Il faut donc plutôt utiliser la fonction sigaction() suivante, plus compliquée à l'usage, mais paramétrable pour ces aspects.

Réception de signaux avec sigaction()

Pour le signal signum, sigaction() installe l'action *new, et renseigne l'ancienne action dans *old si old != NULL :

```
#include <signal.h>
int sigaction(int signum, struct const sigaction * new, struct sigaction * old);
```

Les actions sont des structures de type struct sigaction :

```
struct sigaction {
  sighandler_t sa_handler;  // the handler function for the signal
  int          sa_flags;    // bitwise-OR of SA_RESETHAND, SA_RESTART, etc
  sigset_t     sa_mask;     // set of signals masked during handler execution
};
```

Après exécution de handler, les bits de sa_flags disent si. . .

- SA_RESETHAND : l'action SIG_DFL est réinstallée (*one-shot*),
- SA_RESTART : un appel sytème bloquant interrompu est repris.

Les signaux dans sa_mask seront masqués pendant l'exécution du handler, en plus du signal traité par le handler, qui est tjs masqué. L'ensemble sa_mask se configure avec les fonctions :

```
int sigemptyset (sigset_t * set);             // initialize the empty set
int sigaddset   (sigset_t * set, int signum); // add signum to the set
```

Exemple d'utilisation de sigaction()

Pour un handler *one-shot* + abandon d'appel système bloquant, on peut écrire la fonction oneShotSignal() suivante :

```
void oneShotSignal (int signum, sighandler_t handler) {
  struct sigaction action;
  action.sa_handler= handler;
  action.sa_flags= SA_RESETHAND;  // handler is a one-shot action
  sigemptyset (& action.sa_mask); // no other signal masked except SIGALRM
  sigaction (signum, & action, NULL); // we don't care about the old action
}
```

Si l'on reprend l'exemple installant un handler pour SIGALRM, où signal() est remplacé par oneShotSignal(), on a :

```
bool ALARMED= false; // global variable

void handleAlarm (int signum) {
   (void) signum; // unused, signum is expected to be SIGALRM here
   ALARMED= true;
}

int main (void) {
  oneShotSignal (SIGALRM, handleAlarm);
  ALARMED= false;
  alarm (3); // will interrupt with SIGALRM in about 3 seconds
  ...
  return 0;
}
```