Programmation systèmes avancée
Signaux

Table des matières

1 Introduction **1**

2 Quelques signaux et dispositions par défaut **2**

3 Changement de disposition avec `signal()` **3**

4 Envoyer un signal **4**
4.1 `kill` pour vérifier l'existence d'un processus 4

5 L'ensemble de signaux **4**

6 Le masque de signaux - comment bloquer la réception de signaux **5**
6.1 Les signaux pendants 6
6.2 Le système ne compte pas les occurrences d'un signal pendant 6

7 Attendre un signal **7**

8 Changement de disposition avec `sigaction` **7**

9 Signaux et les appels système **11**

10 Comment écrire les handlers **11**
10.1 Les fonctions « reentrentes » 11
10.1.1 Quelles fonctions qui ne sont pas async-signal-safe ? 12
10.1.2 Fonctions async-signal-safe 12
10.2 Variables globales `sig_atomic_t` 12
10.3 Deux tactiques pour écrire un handler 13

1 Introduction

Un signal est une notification envoyé par le noyau suite à un évènement. Un processus quelconque peut envoyer un signal à un autre processus à condition d'avoir les permissions appropriées.
Quelques exemples d'évènements qui provoquent l'envoie d'un signal par le noyau :

- exception matérielle : instruction machine mal formée, division par 0, l'accès mémoire incorrect (l'adresse incorrecte) ou écriture dans la mémoire accessible uniquement en lecture,
- le signal peut être envoyé suite à une action de l'utilisateur par exemple quand il entre sur le clavier un des caractères de contrôle, par exemple *interrupt* (Ctrl-C), *suspend* (Ctrl-Z),

- évènement software :
  - le signal envoyé quand les données sont disponibles en lecture sur un descripteur,
  - le signal envoyé quand la fenêtre est redimensionnée,
  - la minuterie (timer) envoie un signal `SIGALRM` quand un délais expire,
  - le système envoie le signal `SIGCHLD` au processus parent quand le processus enfant termine ou quand l'enfant passe à l'état STOPPED.

On appelle *disposition* l'action entreprise par un processus à la réception d'un signal. La liste ci-dessous donne les dispositions par défaut à la reception d'un signal par un processus [1] :

1. le signal est ignoré. Le processus destinataire ignore même l'existence de signal.
2. le signal tue le processus (le processus termine à la réception de signal),
3. le fichier *core dump* est généré et le processus termine. Le fichier core dump peut être analysé avec un débogueur pour chercher la source de problème.
4. le processus est arrêté (passe à l'état stopped),
5. l'exécution de processus reprend après un arrêt.

Il est possible de remplacer la disposition par défaut par une autre disposition. Les processus peut installer les dispositions suivantes :

1. ignorer le signal. Le signal ne sera jamais délivré.
2. exécuter un handler. Le processus peut installer une fonction (handler) qui sera exécutée à la réception du signal.
3. reinstaller la disposition par défaut. C'est utile si nous voulons revenir à la disposition par défaut.

Notez qu'il est impossible d'installer la disposition qui termine le processus. Mais il est toujours possible d'installer un handler qui terminera le processus avec `_exit` ou `_Exit`.

2 Quelques signaux et dispositions par défaut

Je ne donnerai pas la liste complète de signaux, au total il y en a une trentaine pour Linux. Pour voir tous les signaux voir *signal(7)* (7 signifie que la page man se trouve dans la section 7, donc pour la voir `man 7 signal`).
Dans la liste ci-dessus, la disposition par défaut pour chaque signal est indiquée entre [] . On utilisera les abréviations suivantes : [term] - terminer le processus, [ign] - ignorer, [core] - terminer le processus et produire le fichier core, [cont] reprendre l'exécution de processus, [stop] - arrêter le processus.
Les noms symboliques des signaux et de toutes les fonctions qui permettent de gérer les signanux sont définis dans le fichier en-tête

```
1 #include <signal.h>
```

`SIGABRT` [core] – le processus qui exécute `abort()` provoque l'envoie de ce signal à soi-même.
`SIGALRM` [term] – le signal généré par le noyau à l'expiration de minuterie enclenchée par `alarm()`,

1. La disposition par défaut dépend de signal, voir la section 2 pour les détails.

`SIGBUS` [core] – « bus error » indique certaines erreurs d'accès à la mémoire, par exemple une référence incorrecte dans la mémoire allouée par `mmap`
`SIGCHLD` [ign] – un processus enfant termine (ou il est arrêté) ,
`SIGCONT` [cont] – redémarre le processus arrêté par SIGSTOP ou SIGTSTP ,
`SIGINT` [term] – le signal envoyé quand l'utilisateur tape sur le clavier le caractère *interrupt* (d'habitude `CTRL-c`). Le pilote du terminal envoie se signal au processus en avant-plan.
`SIGKILL` [term] – termine toujours le processus, ce signal ne peut pas être capté et il est impossible de changer la disposition par défaut
`SIGPIPE` [term] – le signal est généré quand le processus essaie écrire dans un tube (fifo ou pipe) et il n'y a pas de lecteur connecté au tube
`SIGQUIT` [core] – le signal envoyé par le pilote de terminal vers le groupe de processus en avant plan quand l'utilisateur tape le caractère *quit* (d'habitude `CTRL-\`)
`SIGSEGV` [term] – le signal que vous connaissez, sans doute, très bien. Il est généré par un accès invalide à la mémoire (la page mémoire correspondante n'existe pas)
`SIGSTOP` [stop] – met le processus en arrêt. Ce signal ne peut pas être masqué et on ne pourra pas changer l'action par défaut
`SIGTERM` [term] – utilisé pour terminer le processus. Contrairement au `SIGKILL` le signal `SIGTERM` peut être capté et nous pouvons installer un hadler.
`SIGUSR1` et `SIGUSR2` [term] – ces signaux sont réservées pour être utilisées librement dans votre application, le système ne les utilise pas.

3 Changement de disposition avec `signal()`

Un *handler* est une fonction qui est appelée à la réception d'un signal. La fonction `signal` du C standard permet d'installer un handler pour un signal.
Il est plus commode de définir la fonction `signal` en définissant d'abord le type auxiliaire `signalhandler` :

```
1 typedef void (*sighandler)( int );
```

Donc `sighandler` est un pointeur de fonction qui prend l'argument `int` (le signal) et retourne void.
Maintenant nous pouvons définir la fonction `signal` :

```
1 sighandler signal( int sig, sighandler handler)
```

La fonction `signal` prend comme paramètre le numéro de signal et un handler. Elle retourne l'ancien handler. En cas d'erreur la fonction retourne `SIG_ERR`.
Le handler doit être écrit comme une fonction dont la signature est :

```
1 void handler(int sig){
2   /* le code de handler */
3 }
```

Bien sûr le nom de la fonction handler peut être quelconque, ce qui est important c'est la signature : la fonction prend un `int` comme argument et retourne `void`. Si un handler est installé

- à la réception de signal le l'exécution de processus est interrompue,
- la fonction handler est appelée en recevant en paramètre le numéro de signal,

Au retour de la fonction handler le processus reprend l'exécution à l'endroit où il a été interrompu [2]
A la place d'une fonction handler la fonction `signal` peut utiliser les valeurs suivantes :

- `SIG_DFL` – remet la disposition par défaut,
- `SIG_IGN` – indique que le signal doit être ignoré. Le signal ne sera jamais délivré.

La sémantique de `signal` n'est pas assez précise, par exemple il n'est pas clair qu'est-ce qui se passe si pendant l'exécution de handler pour un signal `sig` le même signal `sig` est reçu pour la deuxième fois. Cette nouvelle occurrence de `sig` serait-elle bloquée pendant l'exécution de handler ? Ou bien le handler sera interrompu à son tour et le même handler est relancé de le début ? Ou peut-être la nouvelle instance du signal est tout simplement ignorée ?
Vous devez utilisez toujours la fonction `sigaction` de la section 8 pour installer le handler, elle donne plus de contrôle et a une sémantique plus claire.

4 Envoyer un signal

```
int kill(pid_t pid, int sig)
retourne 0 si OK, -1 si échec
```

`kill` envoie le signal le signal `sig` au processus `pid`.
Le processus doit avoir les permissions appropriées pour envoyer un signal à un autre processus (sinon il pourrait tuer les processus lancés par d'autres utilisateurs ou les processus système !). Si le processus n'a pas de permissions `kill` retourne -1 et `errno==EPERM`.

```
int raise(int sig)
retourne 0 si OK, et -1 sinon
```

permet à un processus d'envoyer un signal à soi-même comme dans `kill(getpid(), sig)`.

4.1 `kill` pour vérifier l'existence d'un processus

Si le paramètre `sig` de `kill` est 0 aucun signal n'est envoyé. `kill` vérifie uniquement si le signal *peut* être envoyé. En particulier si `kill(pid, 0)` retourne -1 et `errno==ESRCH` alors le processus `pid` n'existe pas (même en tant que *zombie*).

5 L'ensemble de signaux

Certaines fonctions utilisent un des signaux représenté par le type `sigset_t`.
Les fonctions suivantes permettent initialiser et tester l'ensemble de signaux :

```
int sigemptyset( sigset_t *set )
int sigfillset( sigset_t *set )
int sigaddset( sigset_t *set, int sig )
int sigdelset( sigset_t *set, int sig )
//ces fonctions retournent 0 is OK, -1 si erreur
```

2. Ce n'est pas exactement vrai si le signal est délivré quand le programme est bloqué sur un appel système bloquant. Dans ce cas l'appel système ne reprend pas mais il est interrompu et retourne une valeur indiquant erreur et la variable `errno` prendra la valeur `EINTR`.

```
6
7 int sigismember(const sigset_t *set, int sig)
8 //retourne 1 si sig appartient à set, 0 sinon, -1 si erreur
```

Les fonctions `sigemptyset` et `sigfillset` servent à initialiser une variable de type `sigset_t` (en passant son adresse) : `sigemptyset` initialise la variable en tant que l'ensemble vide, `sigfillset` initialise `set` comme l'ensemble de tous les signaux.
Une fois la variable `set` initialisée on peut soit ajouter les nouveaux signaux dans `set` avec `sigaddset` soit supprimer des signaux de l'ensemble avec `sigdelset`.
La fonction `sigismember` permet de vérifier si le signal `sig` appartient à l'ensemble `set`.

Exemple. Fabriquer un ensemble de deux signaux, `SIGUSR1` et `SIGUSR2`.

```
1 sigset_t set;
2 sigemptyset( &set );
3 sigaddset( &set, SIGUSR1 );
4 sigaddset( &set, SIGUSR2 );
```

6 Le masque de signaux - comment bloquer la réception de signaux

Pour chaque processus le noyau maintient le masque de signaux. Initialement le masque est vide (aucun signal n'est masqué). Le masque sert à définir l'ensemble de signaux bloqués.
Le signal bloqué n'est pas la même chose que le signal ignoré. Le signal ignoré ne sera jamais délivré, il est perdu pour toujours. Par contre, l'arrivée de signal bloqué est enregistré mais le signal n'est pas délivré tant que ce signal reste bloqué.
La fonction `sigprocmask` sert à modifier le masque de signaux.

```
1 int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)
2 retourne 0 si OK, -1 sinon
```

L'argument `how` détermine l'opération :

- `SIG_BLOCK` : les signaux spécifiés dans l'ensemble `set` sont ajoutés dans le masque,
- `SIG_UNBLOCK` : les signaux spécifiés dans l'ensemble `set` sont enlevés du masque,
- `SIG_SETMASK` : les signaux spécifiés dans `set` remplacent le masque précédent.

`oldset` est l'adresse d'une variable de type `sigset_t` où `sigprocmask` met la précédente valeur du masque. Si on n'est pas intéressé par le masque précédent mettez `NULL` dans `oldset`. Si vous voulez juste récupérer le masque courant mettez `set==NULL`.
Il est impossible de bloquer les signaux `SIGKILL` et `SIGSTOP`. La raison : si le processus pouvait bloquer ces deux signaux alors il aurait été impossible de le tuer ou de l'arrêter !
Pour bloquer tous le signaux (sauf les deux qu'on ne peut pas bloquer) :

```
1 sigset_t set;
2 sigfillset( &set ); /* initialiser avec l'ensemble de tous les signaux */
3 if( sigprocmask( SIG_BLOCK, &set, NULL) == -1 )
4    PANIC_EXIT("sigprocmask");
```

6.1 Les signaux pendants

Le signal envoyé mais pas encore réceptionné par le destinataire est un signal `pendant`.
Quand un processus reçoit un signal qui est actuellement bloqué (masqué) ce signal est ajouté dans l'ensemble de *signaux pendants*.
Quand (et si) le signal pendant est débloqué il est délivré au processus.
La fonction

```
int  sigpending( sigset_t *set )
retourne 0 si OK, -1 si erreur
```

permet de récupérer l'ensemble de signaux pendants.
Exemple. Bloquer le signal `SIGUSR1` et vérifier, après un certain temps, s'il est pendant.

```
sigset_t set;
sigmptyset( &set );
sigaddset( &set, SIGUSR1 );
if( sigprocmask( SIG_BLOCK, &set, NULL ) == -1 )
  PANIC_EXIT("sigprocmask");

/* un code quelconque */

/* trouver l'ensemble de signaux pendants*/
if( sigpending( &set ) == -1 )
  PANIC_EXIT("sigpending");

/* verifier si SIGUSR1 est pendant */
if( sigismember( &set, SIGUSR1 ) ){
    /* SIGUSR1 est  pendant */
}else{
    /* SIGUSR1 non pendant */
}
```

6.2 Le système ne compte pas les occurrences d'un signal pendant

Supposons que le processus P bloque un signal `sig`. Pendant que `sig` reste bloqué il est envoyé 10 fois au processus P. Quand le processus P débloque le signal `sig` il peut être délivré mais le processus P ne recevra qu'une seule instance de `sig` et non pas 10.
Intuitivement, pendant que le signal est bloqué le système mémorise si oui ou non le signal a été envoyé vers le processus mais le système ne compte pas les occurrences du signal.
Donc les signaux peuvent être perdus.
Notez aussi que même les signaux non-bloqués peuvent être perdus.
Supposons qu'on envoie plusieurs fois le même signal `sig` à un processus P qui n'est pas dans l'état « running » mais dans l'état « runnable ». Cela signifie qu'il est prêt à être exécuté mais il attend que le scheduler lui attribue le processeur.
Quand le tour de P arrive le scheduler lui attribue le processeur et P passe à l'état running.
Le signal peut enfin être délivré. Mais le processus P recevra une seule instance de signal `sig`. Comme pour les signaux bloqués, le système enregistre que `sig` a été envoyé au processus P mais le noyau n'enregistre pas combien de fois le signal a été envoyé.

7 Attendre un signal

```
int pause(void)
tjrs retourne -1 et errno==EINTR
```

`pause` suspend l'exécution d'un processus jusqu'à la réception d'un signal (forcement il s'agit de réception d'un signal qui n'est ni masqué ni ignoré).

```
int sigsuspend( const sigset_t *mask )
retourne -1 et errno == EINTR
```

`sigsuspend` remplace le masque courant des signaux par le masque pointé par le paramètre `mask` et suspend l'exécution de processus jusqu'à ce qu'un signal (non-masqué) soit reçu. Le masque courant (celui en vigueur avant l'appel) est sauvegardé par le système. Quand un signal non-masqué est reçu le handler correspondant est exécuté et le masque sauvegardé est restauré.
Donc `sigsuspend()` correspond à l'exécution *atomique* de la suite d'actions :

```
sigprocmask( SIG_SETMASK, &mask, &prevMask );
pause();
sigprocmask( SIG_SETMASK, &prevMask, NULL);
```

La différence entre `sigsuspend()` et cette suite repose sur l'atomicité, `sigsuspend()` bloque les signaux `mask` uniquement pendant l'exécution de handler.
Par contre avec la suite d'actions ci-dessus les signaux restent bloqués même après l'exécution de handler, jusqu'au deuxième appel à `sigprocmask()`.

8 Changement de disposition avec `sigaction`

La fonction `signal` n'est pas très précise en particulier en ce qui concerne le traitement de nouveaux signaux qui arrivent pendant l'exécution de handler.
La fonction `sigaction` permet d'installer un handler et est définie avec plus de précision que `signal`.

```
int sigaction( int sig, const struct sigaction *act,
                        struct sigaction *oldact)
retourne 0 si OK, -1 si erreur
```

Le paramètre `sig` identifie le signal.
`act` est un pointeur vers la structure `struct sigaction`, ou `NULL` si on ne change pas de disposition pour le signal `sig`.
Si `oldsig` est non-`NULL` alors il pointe vers la structure `struct sigaction` qui permet de récupérer l'information concernant l'ancienne disposition.
`struct sigaction` possède les champs suivants :

- `void (*sa_handler)(int)` le pointeur vers un handler à installer pour le signal `sig` . Donc `sa_handler` est soit l'adresse de la fonction handler que vous devez écrire vous-même soit une des constantes `SIG_IGN` ou `SIG_DFL` décrites dans la section 3.
- `sigset_t sa_mask` l'ensemble de signaux bloqués pendant l'exécution de handler. Pendant la durée de l'exécution de handler les signaux de `sa_mask` sont ajoutés

à l'ensemble de signaux bloqués et restent dans l'ensemble de signaux bloqués jusqu'à la fin de l'exécution de handler. Autrement, `sa_mask` permet de spécifier les signaux supplémentaires qui ne peuvent pas interrompre l'exécution de handler. **De plus, par défaut, le signal `sig` qui a provoqué l'exécution de handler est rajouté dans l'ensemble de signaux bloqués.** Cela implique que le handler dont l'exécution est provoquée par le signal `sig` ne sera lui-même pas interrompu par une nouvelle instance du même signal `sig`.

- `int sa_flags` – les flags qui contrôlent l'appel au handler.

Le champ `sa_flags` est un OR bit-à-bit de constantes :

- `SA_NOCLDSTOP` : Utilisé uniquement quand `sig == SIGCHLD`. Le signal `SIGCHLID` sera généré uniquement quand le processus enfant termine [3].
- `SA_NOCLDWAIT` : Utilisé uniquement avec `sig == SIGCHLD`. Quand ce flag est spécifié un processus enfant qui termine ne sera pas transformé en zombie. Notez que c'est une manière de ne pas avoir des enfants zombie qui permet de s'affranchir de l'utilisation de `wait` ou `waitpid`.
- `SA_NODEFER` : Rappelons que par défaut pendant l'exécution de handler sur un signal `sig`, le signal `sig` est masqué. La constante `SA_NODEFER` indique que nous ne voulons plus que ce signal soit masqué. Avec cette constante l'exécution du handler peut être interrompu par le même signal qui est traité par le handler.
- `SA_RESETHAND` Après l'exécution de handler le hadler sera supprimé et on revient à la disposition par défaut pour le signal `sig`. Intuitivement cela signifie que le handler pour `sig` est installé pour être exécuté une seule fois.
- `SA_RESTART` On demande qu'un appel système interrompu par le signal soit automatiquement repris, voir la section 9.

Exemple.
Installer le handler qui compte le nombre de signaux `SIGUSR1` reçus par le processus.
Installer un autre handler pour le signal `SIGTERM` qui affiche le nombre de signaux `SIGUSR1` reçus et qui termine le processus.
Tous les signaux doivent être bloqués pendant l'exécution du premier handler [4]. Les signaux `SIGUSR1` et `SIGTERM` doivent être bloqués pendant l'exécution du deuxième handler.

```
/* le compteur */
static unsigned long count = 0;

/* les handlers */
static void handler_compteur(int sig){
  count ++;
  return;
}

static void handler_terminate(int sig){
  printf( "# signaux recus %ld\n", count);
   _exit(0);
}

```

3. Rappelons que par défaut le signal `SIGCHLD` est envoyé pour trois raisons : (1) le processus enfant termine, (2) le processus enfant est arrêté, (3) le processus enfant redémarre après l'arrêt.
4. sauf bien sûr ceux qui ne peuvent jamais être bloqués, voir la section 6.

```c
int main(void){

  struct sigaction act;

  /*  la fonction handler */
  act.sa_handler = handler_compteur;

  /* bloquer tous les signaux pendant l'execution de
   * handler  pour SIGUSR1                        */
  sigfillset( &act.sa_mask ) ;

  /* sa_flags non-utilises, initialiser a 0*/
  act.sa_flags = 0;

  /* installer le handler pour le signal SIGUSR1  */
  if( sigaction( SIGUSR1 , &act, NULL) == -1 )
    exit_err("sigaction");

  /* installer le deuxieme handler */
  /*  la fonction handler */
  act.sa_handler = handler_terminate;

  /* bloquer SIGUSR1 pendant l'execution de handler  */
  sigemptyset( &act.sa_mask ) ;
  sigaddset( &act.sa_mask, SIGUSR1);

  /* sa_flags non-utilises, initialiser a 0*/
  act.sa_flags = 0;

  /* installer le handler pour le signal SIGUSR1  */
  if( sigaction( SIGTERM , &act, NULL) == -1 )
    exit_err("sigaction");

  while( 1 ){
    sleep(3); /* simuler le travail par sleep()*/
  }
  return 0;
}
```

Dans l'exemple ci-dessus le handler `handler_terminate` fait l'appel à la fonction `printf`. Cette fonction n'est pas *async-synch-safe* (voir la section 10.1.1) donc **elle ne doit pas être utilisée** dans un handler. Mais tant que le programme principale (ici juste `main()`) n'utilise aucune fonction de `stdio.h` il n'y a pas danger d'interférence avec `printf()` dans le handler.
Ceci dit il est très fortement recommandé de se limiter aux fonctions async-signal-safe dans les handlers.
Une possibilité à palier au problème de `printf` dans le handler c'est le remplacer par `write` qui est async-signal-safe.
Et finalement, au lieu de terminer le processus par un appel à `_exit` dans le handler `handler_terminate` on peut utiliser le flag global (variable `volatile sig_atomic_t` décrite

dans la section 10.2) comme ci-dessous :

```c
volatile  sig_atomic_t flag = 0;

static unsigned long count = 0;

static void handler_compteur(int sig){
  count ++;
  return;
}

static void handler_terminate(int sig){
  flag = 1;
  return;
}

int main(void){

  struct sigaction act;

  act.sa_handler = handler_compteur;

  sigfillset( &act.sa_mask ) ;

  act.sa_flags = 0;

  if( sigaction( SIGUSR1 , &act, NULL) == -1 )
    PANIC_EXIT("sigaction");

  act.sa_handler = handler_terminate;

  sigemptyset( &act.sa_mask ) ;
  sigaddset( &act.sa_mask, SIGUSR1);

  act.sa_flags = 0;

  /* installer le handler pour le signal SIGUSR1  */
  if( sigaction( SIGTERM , &act, NULL) == -1 )
    PANIC_EXIT("sigaction");

  while( 1 ){
    sleep(3); /* simuler le travail par sleep()*/
    if( flag == 1 }{ /* verifier le  flag */
      printf("nombre de signaux %s recus = %lu\n", argv[1], count);
      exit(0);
    }
  }
  return 0;
}
```

Dans cette solution `printf` a été déplacé à l'extérieur du handler mais maintenant il faut pério-

diquement vérifier si `flag` passe à 1. Donc il y a une sorte d'attente active.

9 Signaux et les appels système

Supposons qu'un signal est reçu quand le processus attend les données (avec `read`) sur un descripteur ouvert sur une tube (pipe ou fifo). L'appel système `read` sera interrompu, le handler exécuté et `read` retournera avec erreur et `errno == EINTR` (`EINTR` est la valeur erreur utilisée pour tout appel système interrompu par un signal, pas seulement pour `read`).
Pour reprendre le `read` interrompu par un signal il faudra utiliser le code suivant :

```
while( ( cnt = read(fd, buf, BUF_SIZE ) == -1 && errno == EINTR )
   continue;
if( cnt == -1 )
   PANIC_EXIT("read"); /* autre erreur */
```

Au lieu de faire un boucle qui vérifie si `errno == EINTR` on peut utiliser le flag `SA_RESTART` à l'installation de handler par `sigaction`. Cette flag assure que l'appel système interrompu par le signal reprenne automatiquement au retour du handler. (Hélas cela n'est pas vrai pour tous les appels système).

10 Comment écrire les handlers

Les handlers doivent être les plus simples possible. Il y a deux façons habituelles d'écrire un handler :

(1) le handler met à jours quelques variables globales et retourne. Le programme principale teste périodiquement ces variables pour prendre une action appropriée.

(2) la solution précédente n'est pas applicable au processus qui attend les données sur des descripteurs (descripteurs de fifo, pipe ou socket). Il ne peut pas vérifier l'état des variables globales puisque il est bloqué sur un appel à `select` ou `poll`. Le processus doit créer un pipe et ajouter le descripteur de lecture du pipe dans l'ensemble de descripteurs de `select` ou `pipe`. Dans le handler le processus écrit un octet dans le pipe et au retour de `select` ou `poll` vérifie si le pipe contient ce octet. Si c'est le cas cela signifie que hanler a été exécuté et que le processus a reçu le signal.

   Notez d'habitude les pipes sont utilisés pour assurer la communication entre un processus et ces descendants tandis qu'ici, pour la première fois, le pipe est utilisé pour la communication avec le processus lui-même (intuitivement pour la « communication » entre le handler et le code principal du processus).

(3) le handler effectue une sorte de nettoyage (vide les tampons, supprime les fichiers temporaires) et soit termine le processus soit fait un saut non-local dans le programme principal (les sauts non-locaux ne sont pas couverts par ce cours).

10.1 Les fonctions « reentrentes »

Intuitivement, une fonction est reentrante si elle peut être exécutée sans danger en parallèle par plusieurs threads. Quel est le rapport avec les handlers ?
L'exécution de handler interrompt l'exécution de programme principal, cette interruption peut avoir lieu à n'importe quel moment, par exemple au milieu de l'exécution d'une fonction et c'est la fonction de handler qui prend la main.

Mais c'est exactement la même situation quand on passe d'un thread à un autre dans un processus. Les fonctions qui peuvent être utilisées en parallèle par plusieurs threads doivent être reentrentes.
Dans le cadre de handlers on ne parle pas de fonctions reentrantes mais des fonctions *async-signal-safe*. Cette notion est plus large que la notions de fonctions reentrantes.
Une fonction peut être async-signal-safe parce qu'elle bloque les signaux pendant toute son exécution.
Pour être sûr que les handlers soient corrects dans les handlers il faut utiliser uniquement les fonctions async-signal-safe.

10.1.1 Quelles fonctions qui ne sont pas async-signal-safe ?

Toute fonction qui modifie les données globales ou `static` n'est pas async-signal-safe. Si elle est interrompue par un handler qui fait appel à la même fonction les données peuvent se retrouver dans un état inconsistant. L'appel d'une fonction qui n'est async-signal-safe dans un handler peut terminer le processus de façon inattendue et imprévisible.
Les fonctions `malloc`, `free` et compagnie **ne sont pas** async-signal-safe. Elles manipulent en interne les listes chaînées de blocs de mémoire libres et occupés.
Toutes les fonction de la bibliothèque standard `stdio.h` d'entrées/sorties, `printf`, `fprintf` `scanf`, `fgetc`, `fread`, `fwrite`, `fputc` ... etc., **ne sont pas async-signal-safe**. Elles utilisent le tampon (buffer) (caché derrière le type `FILE`) donc si une de ces fonctions est interrompue et le handler fait appel à une fonction pour le même flot `FILE` le résultat est imprévisible.

10.1.2 Fonctions async-signal-safe

Le nombre de fonctions *async-signal-safe* est limité. Ce sont les seules fonctions que vous devez utiliser sans danger dans un handler et qui garantissent le comportement correct du programme.
Les fonctions suivantes sont async-signal-safe [5] :

- `open`, `close`, `read`, `write`, `unlink`, `dup`, `dup2`, `fcntl`, `fstat`, `stat`, `lseek`, `ftruncate`, `pipe`, `unlink`
- `execle`, `execve`, `fork`, `wait`, `waitpid`, `getpid`, `getppid`, `sleep`, `_exit`, `_Exit`
- `sigaction`, `sigaddset`, `sigdelset`, `sigemptyset`, `sigfillset`, `sigpause`, `sigsuspend`, `signal`.

Notez, comme j'ai indiqué dans la section précédente, les fonctions de haut niveau d'entrées/-sorties ne sont pas async-signal-safe, par contre les fonctions de bas niveau qui opèrent sur les descripteurs de fichiers peuvent être utilisées dans les handlers.
Notez aussi que la fonction `exit()` n'est pas async-signal-safe par contre `_exit()` est async-signal-safe.
La liste complète de fonction async-signal-safe se trouve sur la page `man` de `sigaction`.

10.2 Variables globales `sig_atomic_t`

Souvent nous voulons que le programme principale et le handler partagent les données globales. Le problème est que les opérations sur les données globales ne sont pas atomiques, même lire et écrire une variable `int` n'est pas une opération atomique mais peut se traduire en plusieurs instructions machine. Le signal peut interrompre le programme principale au milieu de l'opération de lecture ou écriture d'une variable. Donc à priori il n'est pas garantie qu'une variable globale partagée par un handler et le programme principal soit dans un état consistent.

5. cette liste n'est pas complète

Pour cette raison C introduit un nouveau type entier `sig_atomic_t`. Le système garantie que la lecture de la valeur de la variable `sig_atomic_t` et l'affectation d'une nouvelle valeur à `sig_atomic_t` sont des opérations atomiques. Cela implique que le handler et le programme principal peuvent utiliser une variable globale déclarée comme

```
1 volatile sig_atomic_t flag;
```

L'adjectif `volatile` indique au compilateur de ne pas faire optimisation d'accès à cette variable [6] Notez que même pour une variable `volatile sig_atomic_t flag` les opérations `flag++` et `flag--` **ne sont pas atomiques**. C'est qui est atomique c'est l'affectation simple comme `flag = 4` et le test comme `if( flag == 2 )` .
Les constantes `SIG_ATOMIC_MIN` et `SIG_ATOMIC_MAX` définie dans `stdint.h` donnent les valeurs minimale et maximale de `sig_atomic_t` (C ne spécifie pas si `sig_atomic_t` est un entier signé ou non).

10.3 Deux tactiques pour écrire un handler

Il y a deux possibilité pour écrire un handler correct :

- soit il faut s'assurer que le code de handler soit reentrant et que les fonctions utilisées dans le handler soient async-signal-safe,
- soit bloquer tous les signaux dans le programme principal au moment où ce programme utilise les fonctions qui ne sont pas async-signal-safe.

6. Sans `volatile` le compilateur qui optimise le code peut déduire incorrectement la valeur de la variable `sig_atomic_t`. L'ajout de `volatile` indique au compilateur de ne pas essayer de déduire la valeur de la variable mais toujours aller chercher la valeur dans la variable elle-même.