كلية العلوم – القنيطرة
Faculté des Sciences – Kenitra

Cours du module

Programmation Système

SMI S6

Prof. ADIL EL MAKRANI

2019-2020
Département Informatique

Sommaire

Ch1. Fonctions des processus

1. Introduction

Nous allons commencer notre étude de la programmation en C sous Linux par plusieurs chapitres analysant les divers aspects de l'exécution des applications. Ce chapitre introduira la notion de processus. Ainsi que les différents identifiants qui y sont associés, leurs significations et leurs utilisations dans le système.

Dans les chapitres suivants, nous étudierons les interactions qu'un processus peut établir avec son environnement, c'est-à-dire sa propre vision du système, configurée par l'utilisateur, puis l'exécution et la fin des processus. En analysant toutes les méthodes permettant de démarrer un autre programme, de suivre ou de contrôler l'utilisation des ressources, de détecter et de gérer les erreurs.

2. Présentation des processus

Sous Unix, toute tâche qui est en cours d'exécution est représentée par un processus. Un processus est une entité comportant à la fois des données et du code. On peut considérer un processus comme une unité élémentaire en ce qui concerne l'activité sur le système.

Puisque le processeur ne peut exécuter qu'une seule instruction à la fois, le noyau va donc découper le temps en tranches de quelques centaines de millisecondes (quantum de temps) et attribuer chaque quantum à un programme

➔ Le processeur bascule entre les programmes. Et l'utilisateur voit ses programmes s'exécuter en même temps. Pour l'utilisateur, tout se passe comme si on a une exécution réellement en parallèle.

2.1 Définitions

Programme : c'est un fichier exécutable stocké sur une mémoire de masse. Pour son exécution, il est chargé en mémoire centrale.

Processus, est un concept central dans tous système d'exploitation :

C’est un programme en cours d’exécution ; c’est-à-dire, un programme à l’état actif. C’est l’image de l’état du processeur et de la mémoire pendant l’exécution du programme. C’est donc l'état de la machine à un instant donné.

2.2 Relations entre processus

Compétition :

Situation dans laquelle plusieurs processus doivent utiliser simultanément une ressource à accès exclusif

Exemples : imprimante

Coopération :

Situation dans laquelle plusieurs processus collaborent à une tâche commune et doivent se synchroniser pour réaliser cette tâche.

Synchronisation : La synchronisation se ramène au cas suivant :

Un processus doit attendre qu’un autre processus ait terminé.

2.3 États d’un processus

Lorsqu'un processus n'a pas toutes les ressources dont il a besoin pour s'exécuter, il est nécessaire de le bloquer en attendant que ces ressources soient disponibles. La figure ci-dessous montre les différents états que peut prendre un processus, lorsqu'il existe. Lors de la création il est mis, en général, dans l'état bloqué, en attendant qu'il ait toutes les ressources dont il a besoin initialement. Sa destruction peut subvenir dans n'importe quel état à la suite d'une décision interne s'il est actif, ou externe s'il est dans un autre état. Dans ce cas, il faut récupérer toutes les ressources qu'il possédait

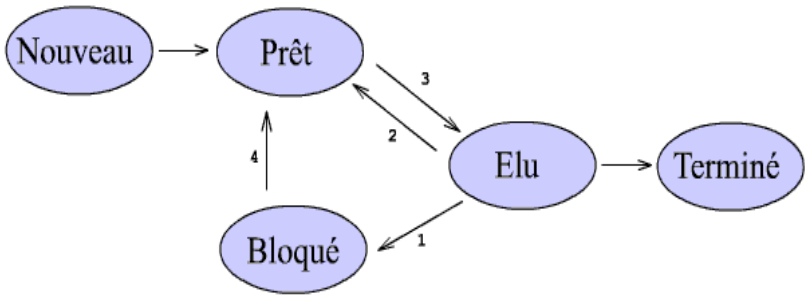


Figure : Les principaux états d'un processus.

Actif (Elu, en Exécution) : Le processus en cours d’exécution sur un processeur. On peut voir le processus en cours d’exécution en tapant la commande **« *ps* »** sur une machine Unix.

Un processus **élu** peut-être arrêter, même s'il peut poursuivre son exécution. Le passage de l'état actif à l'état prêt est déclenché par le noyau lorsque la tranche de temps attribué au processus est épuisée.

Prêt : Le processus est suspendu provisoirement pour permettre l'exécution d'un autre processus. Le processus peut devenir actif dès que le processeur lui sera attribué par le système.

Bloqué (en attente) : Le processus attend un événement extérieur (une ressource) pour pouvoir continuer, par exemple lire une donnée au clavier. Lorsque la ressource est disponible, il passe à l'état "prêt".

Terminé : le processus a terminé son exécution.

3. Identification par le PID

Chaque programme (fichier exécutable ou script Shell, Perl) en cours d'exécution dans le système correspond à un (ou parfois plusieurs) processus du système. Chaque processus possède un numéro de processus (PID).

Sous Unix, on peut voir la liste des processus en cours d'exécution, ainsi que leur PID, par la commande **ps**, qui comporte différentes options.

Pour voir ses propres processus en cours d'exécution on peut utiliser la commande

```
$ ps x
```

Pour voir l'ensemble des processus du système, on peut utiliser la commande

```
$ ps −aux
```

Pour voir l'ensemble des attributs des processus, on peut utiliser l'option -f. Par exemple, pour l'ensemble des attributs de ses propres processus, on peut utiliser

```
$ ps −f x
```

Un programme C peut accéder au PID de son instance en cours d'exécution par la fonction **getpid**(), qui retourne le PID :

```
pid_t getpid (\ void ) ;
```

Chaque processus est identifié par un numéro unique, le PID (Processus IDentification) et il appartient à un propriétaire identifié par UID (User ID) et à un groupe identifié par GID

(Group ID). Le PPID (Parent PID) d'un processus correspond au PID du processus qui l'a créé.

La fonction « **int getpid()** » renvoie le numéro **(PID)** du processus en cours.

La fonction « **int getppid()** » renvoie le numéro du processus père **(PPID).** Chaque processus a un père, celui qui l'a créé.

La fonction « **int getuid()** » permet d'obtenir le numéro d'utilisateur du processus en cours **(UID)**.

La fonction « **int getgid()** » renvoie le numéro du groupe du processus en cours **(GID).**

4. Création d'un processus

Le premier processus du système, init, est créé directement par le noyau au démarrage.

La seule manière, ensuite, de créer un nouveau processus est d'appeler l'appel-système **fork( ),** qui va dupliquer le processus appelant. Au retour de cet appel-système, deux processus identiques continueront d'exécuter le code à la suite de **fork( ).** La différence essentielle entre ces deux processus est un numéro d'identification. On distingue ainsi le processus original, qu'on nomme traditionnellement le processus père, et la nouvelle copie.

Le processus fils. L'appel-système fork( ) est déclaré dans <unistd.h>, ainsi :

pid_t fork(void);

La fonction « fork() » renvoit un entier.

Le « pid_t » est nouveau type qui est identique à un entier. Il est déclaré dans «/sys/types.h». Il est défini pour l'identificateur du processus. A la place de «pid_t» on peut utiliser «int».

Le processus qui a invoqué la fonction « **fork()** » est appelé le processus père tandis que le processus créé est appelé le processus fils.

Le père et le fils ne se distinguent que par la valeur de retour de «**fork()**».

Dans le processus père: la fonction « **fork()** » renvoie le numéro du processus nouvellement créé (le processus fils).

Dans le processus fils : la fonction « **fork()** » renvoie **0**.

En cas d'échec, le processus fils n'est pas crée et la fonction renvoie **« -1 ».**

4.1 Utilisation classique sans gestion des erreurs :

```
# include <unistd.h>
# include <stdio.h>
...
if (fork() != 0) {   /*Exécution du code correspondant au processus père */ }
else {        /* if (fork() == 0) */
/*Exécution du code correspondant au processus fils */
}
```

Exemple 4.1: Le processus père crée un fils, ensuite chaque processus affiche son identifiant.

```
#include <stdio.h>
#include <unistd.h>
int main() {
if(fork() !=0) {
printf("Je suis le pere: ");
printf(" Mon PID est %d \n",getpid());
} else {
printf("Je suis le fils:");
printf(" Mon PID est %d\n",getpid());
} }
```

Exécution:

% testfork

Je suis le pere: Mon PID est 1430

Je suis le fils: Mon PID est 1431

%

Attention: l'affichage peut apparaître dans l'ordre inverse.

Exemple 4.2: Le processus père crée un fils ensuite affiche son identifiant ainsi que celui de son fils. Le fils affiche son identifiant ainsi que celui de son père.

```
#include <stdio.h>
#include <unistd.h>
int main() {
pid_t pid=fork(); // appel de la fonction fork()
if (pid!=0) {
printf("Je suis le pere:");
printf(" mon PID est %d et le PID de mon fils est %d \n", getpid(), pid);
} else {
```

Exemple d'exécution:

% testfork

Je suis le pere: Mon PID est 1430 et le PID de mon fils est 1431

Je suis le fils: Mon PID est 1431 et le PID de mon pere est 1430

```
printf("Je suis le fils.");
printf(" Mon pid est:%d et le PID de mon pere est %d \n ",getpid(), getppid());
} }
```

Le processus créé (le processus fils) est un clone (copie conforme) du processus créateur (père). Il hérite de son père le code, les données la pile et tous les fichiers ouverts par le père. Mais attention : les variables ne sont pas partagées. Un seul programme, deux processus, deux mémoires virtuelles, deux jeux de données

Exemple 4.3

```
int i=10 ;
if (fork() != 0) {
printf("Je suis le père, mon PID est %d\n", getpid());
i+= 2;  }
else {
printf("Je suis le fils, mon PID est %d\n" , getpid());
i+=5; }
printf("Pour PID = %d, i = %d\n", getpid(), i);
}
```

Exemple d'exécution

Je suis le fils, mon PID est 1361

Je suis le père, mon PID est 1360

Pour PID=1360, i = 12

Pour PID=1361, i = 15

4.2 Appel système de « fork() » avec gestion des erreurs

Rappel sur la variable « errno »

Dans le cas où une fonction renvoie « **-1** » c'est qu'il y a eu une erreur, le code de cette erreur est placé dans la variable globale « errno », déclarée dans le fichier « errno.h ». Pour utiliser cette variable, il faut inclure le fichier d'en-tête (#include <errno.h>).

Rappel sur la fonction « perror() »

La fonction « perror() » renvoie une chaine de caractères décrivant l'erreur dont le code est stocké dans la variable « errno ».

Syntaxe de la fonction « perror() » void perror(const char *s);

- La chaine « s » sera placée avant la description de l'erreur. La fonction affiche la chaine « s » suivie de deux points, un espace blanc et en fin la description de l'erreur et un retour à la ligne

Attention: Il faut utiliser « perror() » directement après l'erreur.

Exemple 4.5 : Utilisation classique de « fork() » avec gestion des erreurs:

```
# include <unistd.h> # include <stdio.h> #include <errno.h>
if (fork() == - 1) {       /* code si échec \n */
printf ("Code de l'erreur pour la création: %d \n", errno);
perror("Erreur de création ");
}
else
if (fork() != 0) {   /* Exécution du code correspondant au processus père */ }
else { /* if (fork() == 0) */
/* Exécution du code correspondant au processus fils */
} }
```

5. Primitives de recouvrement : les primitives exec()

Il s'agit d'une famille de primitives permettant le lancement de l'exécution d'un programme externe. Il n'y a pas création d'un nouveau processus, mais simplement changement de programme. Il y a six primitives **exec()** que l'on peut répartir dans deux groupes : les **execl(),** pour lesquels le nombre des arguments du programme lancé est connu, puis les **execv()** où il ne l'est pas. En outre toutes ces primitives se distinguent par le type et le nombre de paramètres passés.

Premier groupe d'exec(). Les arguments sont passés sous forme de liste :

*int execl(char *path, char *arg0, char *arg1,..., char *argn,NULL)*

/* exécute un programme */

/* path : chemin du fichier programme */

/* arg0 : premier argument */

/* ... */

/* argn : (n+1) ième argument */

Dans **execl** , **path** est une chaîne indiquant le chemin exact d'un programme. Un exemple est "/bin/ls".

La primitive «execlp() »

*int execlp(char *fiche, char *arg0,char *arg1,...,char *argn, NULL)*

Dans **execlp**, le " **p** " correspond à " **path** " et signifie que les chemins de recherche de l'environnement sont utilisés. Par conséquent, il n'est plus nécessaire d'indiquer le chemin complet. Le premier paramètre de **execlp** pourra par exemple être "**ls**". Le second paramètre des trois fonctions **exec** est le nom de la commande lancée et reprend donc une partie du premier paramètre. Si le premier paramètre est "/**bin/ls**", le second doit être "**ls**". Pour la troisième commande, le second paramètre est en général identique au premier si aucun chemin explicite n'a été donné.

Exemple 5.1: lancer la commande « ls -l /tmp » à partir d'un programme.

```
#include <stdio.h> #include <unistd.h>
int main(void){
execl("/usr/bin/ls","ls","-l","/tmp", NULL) ;
perror("echec de execl \n");
}
```

Exemple 5.2 : lancer la commande « ls -l /tmp » à partir d'un programme.

```
#include <stdio.h> #include <unistd.h>
int main(void){
execlp("ls","ls","-l","/tmp", NULL) ;
perror("echec de execlp \n");
}
```

Cas où les arguments sont passés sous forme de tableau

Primitive « execv() » : int execv(char *path, char *argv[])

Primitive « execvp() » :int execvp(char *fiche,char *argv[])

« argv » : pointeur vers un tableau qui contient le nom et les arguments du programme à exécuter. La dernière case du tableau vaut « NULL ».

« path » et « fiche» ont la même signification que dans les primitives précédentes

6. Primitive sleep ()

La fonction **sleep ()** suspend un processus pendant le nombre de secondes indiqué en paramètre. La fonction peut être interrompue par un signal (comme on le verra plus loin dans le cours) ; dans ce cas, la fonction fournit le nombre de secondes restant avant la fin programmée, 0 sinon.

La syntaxe est : unsigned int sleep (unsigned int seconds)

Exemple 6.1 : Le programme suivant bloque le processus père de 10 secondes (appel de « sleep(10) ») et le processus fils de 5 secondes (appel de « sleep(5) ») .

```
#include <stdio.h> # include <stdlib.h>#include <unistd.h>
int main() {
if (fork() != 0) {
printf(" je suis le père, mon PID est %d\n", getpid());
sleep(10)
/* le processus père est bloqué pendant 10 secondes */
} else {
printf(“ je suis le fils, mon PID est %d\n", getpid());
sleep(5)
 /* le processus fils est bloqué pendant 5 secondes */       }
printf(" PID %d : Terminé \n", getpid())       }
```

Compilation: gcc -o testfork testfork.c

On lance « testfork » en backround, ensuite en lance la commande « ps -f »

Exécution: ./testfork & ps -f

```
je suis le fils, mon PID est 2436
je suis le père, mon PID est 2434
[2] 2434
UID    PID    PPID   TTY    .... CMD
mak    1816   1814   pts/0  .... bash
mak    2434   1816   pts/0  ..... ./testfork
mak    2435   1816   pts/0  ..... ps -f
mak    2436   2434   pts/0  ..... ./testfork
2436 : Terminé
2434 : Terminé
```

7. La primitive «exit() »

La primitive «**exit()** » permet de terminer le processus qui l'appelle. Elle est déclarée dans le fichier « **stdlib.h** ».

Pour utiliser cette primitive, il faut inclure le fichier d'entête « stdlib.h » **(#include <stdlib.h>)**.

Syntaxe : void exit (**int status**)

« **status** » est un code de fin qui permet d'indiquer au processus père comment le processus fils a terminé (par convention: **status=0** indique une terminaison normale sinon indique une erreur).

Tous les descripteurs de fichier ouverts sont fermés. Si le père meurt avant ses fils, le père du processus fils devient le processus init de pid 1. Par convention, un code de retour égal à zéro signifie que le processus s'est terminé normalement, et un code non nul (généralement 1 ou -1) signifie qu'une erreur s'est produite.

Exemple 7.1

```
#include <stdio.h> # include <stdlib.h> #include <unistd.h>
main() {
if (fork() == 0) { // fils
printf("je suis le fils, mon PID est %d\n", getpid());
exit(0);
/* la partie qui suit est non exécutable par le fils */
} else {
printf("je suis le pere, mon PID est %d\n", getpid());
} }
```

8. Primitive wait ()

L'appel système wait () bloque un processus en attente de la fin de l'exécution d'un fils qu'il a crée. La syntaxe est la suivante : pid_t wait (int* status) ; La fonction retourne le numéro de PID du fils qui vient de se terminer ; status peut recevoir des informations s'il est non NULL sur la façon dont s'est terminé le processus (valeur du exit du fils). Si on a

créé plusieurs fils, l'appel wait ne permet pas d'attendre la fin d'un fils particulier. C'est le rôle de l'appel système waitpid

Une première synchronisation est réalisable par l'intermédiaire de la commande *wait*. Celle-ci a pour effet de bloquer l'exécution du processus en attendant le mort d'un de ses processus fils. Soit le code suivant :

```
int main() {
/* partie de commune */
int n=0 ;
if (fork() ==0)
{ /* processus fils*/
 printf("n=%d\n" , n+2) ; }
 else { /* processus père */
wait(NULL) ;
printf("n=%d\n" , n+1) ; } }
```

Ce programme affichera forcément 2 puis 1 car la commande *wait* placée au début du processus père force ce processus à attendre la fin du processus fils. Le +2 s'exécute alors en premier. Une fois l'affichage de 2 réalisé, le fils meurt. Il provoque alors le déblocage de la commande *wait*. Le +1 peut alors se réaliser et l'affichage de 1 aussi.

Chap. II : Gestion des signaux sous Unix

1. Généralités

La gestion des signaux entre processus est peut-être la partie la plus passionnante de la programmation sous Unix. C'est aussi celle qui peut conduire aux dysfonctionnements les plus subtils, avec des bogues très difficiles à détecter de par leur nature fondamentalement intempestive.

Le principe est a priori simple : un processus peut envoyer sous certaines conditions un signal à un autre processus (ou à lui-même). Un signal peut être imaginé comme une sorte d'impulsion qui oblige le processus cible à prendre immédiatement (aux délais dus à l'ordonnancement près) une mesure spécifique.

Le destinataire peut soit ignorer le signal. Soit le capturer c'est-à-dire dérouter provisoirement son exécution vers une routine particulière qu'on nomme gestionnaire de signal (signal handler). Soit laisser le système traiter le signal avec un comportement par défaut.

Un signal peut être envoyé par :

Le système d'exploitation (déroutement : événement intérieur au processus généré par le hard) pour informer les processus utilisateurs d'un événement anormal (une erreur) qui vient de se produire durant l'exécution d'un programme.

Un processus utilisateur : pour se coordonner avec les autres, par exemple pour gérer une exécution multiprocessus plus ou moins complexe

2. Caractéristiques des signaux

Il existe un nombre « **NSIG** » ou « **_NSIG**» signaux différents. Ces constantes sont définies dans « **signal.h** ».

Chaque signal est identifié par

Un nom défini sous forme de constante symbolique commençant par « SIG ». Les noms sont définis dans le fichier d'en-tête <**signal.h**>

Un numéro, allant de 1 à « NSIG ».

La commande **kill –l** permet d'afficher la liste de tous les signaux définis dans le système.

- On ne connaît pas l'émetteur du signal.
- Le signal « 0 » a un rôle particulier. On l'utilise pour vérifier l'existence d'un signal.

2.1 Etats des signaux

Un signal pendant (**pending**) est un signal en attente d'être pris en compte.

Un signal est délivré lorsqu'il est pris en compte par le processus qui le reçoit.

La prise en compte d'un signal entraîne l'exécution d'une fonction spécifique appelée **handler**, c'est l'action associée au signal. Elle peut être soit :

La fonction prédéfinie dans le système (action par défaut).

Une fonction définie par l'utilisateur pour personnaliser le traitement du signal.

2.2 Liste incomplète des signaux

SIGABRT : terminaison anormale du processus.

SIGALRM : alarme horloge: expiration de timer

SIGFPE : erreur arithmétique

SIGHUP : rupture de connexion

SIGILL : instruction illégale

SIGINT : interruption terminal

SIGKILL : terminaison impérative. Ne peut être ignoré ou intercepter

SIGPIPE : écriture dans un conduit sans lecteur disponible

SIGQUIT : signal quitter du terminal

SIGSEGV : accès mémoire invalide

SIGTERM : signal 'terminer' du terminal

SIGUSR1 : signal utilisateur 1

SIGUSR2 : signal utilisateur 2

SIGCHLD : processus fils stoppé ou terminé

SIGCONT : continuer une exécution interrompue

SIGSTOP : interrompre l'exécution. Ne peut être ignoré ou intercepter

SIGTSTP : signal d'arrêt d'exécution généré par le terminal

SIGTTIN : processus en arrière plan essayant de lire le terminal

SIGTTOU : processus en arrière plan essayant d'écrire sur le terminal

SIGBUS : erreur accès bus

SIGPOLL : événement interrogeable

SIGPROF : expiration de l'échéancier de profilage

SIGSYS : appel système invalide

SIGTRAP : point d'arrêt exécution pas à pas

SIGURG : donnée disponible à un socket avec bande passante élevée

SIGVTALRM : échéancier virtuel expiré

SIGXCPU : quota de temps CPU dépassé

SIGXFSZ : taille maximale de fichier dépassée

3. Les différents traitements par défaut des signaux

A chaque type de signal est associé un gestionnaire du signal (« handler ») par défaut appelé « SIG_DFL ». Les 5 traitements par défaut disponibles sont :

- Terminaison normale du processus.
- Terminaison anormale du processus (par exemple avec fichier core).
- Signal ignoré (signal sans effet).
- Stoppe le processus (le processus est suspendu).
- Continuation d'un processus stoppé.

Un processus peut ignorer un signal en lui associant le handler «SIG_IGN ».

« SIG_DFL » et « SIG_IGN » sont les deux seules macros prédéfinies.

3.1 Signaux particuliers

Pour tous les signaux, l'utilisateur peut remplacer le handler par défaut par un handler personnalisé qui sera défini dans son programme, à l'exception de certains signaux qui ont des statuts particuliers et qui ne peuvent pas être interceptés, bloqués ou ignorés :

- « SIGKILL » permet de tuer un processus.
- « SIGSTOP » permet de stopper un processus (stopper pour reprendre plus tard, pas arrêter).
- « SIGCONT » permet de faire reprendre l'exécution d'un processus stoppé (après un « SIGSTOP »).

3.2 Manipulation des signaux

Un processus peut envoyer un signal à un autre processus ou à un groupe de processus. Un processus qui reçoit le signal agit en fonction de l'action (handler) associé au signal. On peut mettre, à la place des handlers définis par défaut, des handlers particuliers permettant un traitement personnalisé. La manipulation des signaux peut se faire:

- Par la ligne de commande (frappe au clavier). Par exemple des combinaisons des certains caractères tapés au clavier:
  Interruption (Ctl-c) -> SIGINT
  Terminaison (Ctl-d) -> SIGQUIT
  Suspension (Ctl-z) -> SIGTSTP
- Dans un programme utilisateur, essentiellement par les deux primitives principales : la fonction « **kill()** » pour envoyer des signaux. Ou les fonctions « **signal()** » ou « **sigaction()** » pour mettre en place une action personnalisée à un signal donné.

4. Envoi d'un signal par un processus

La méthode la plus générale pour envoyer un signal est d'utiliser soit la commande sheel kill(1), soit la fonction C kill(2).

4.1 La commande kill

La commande kill prend une option -signal et un pid.

Exemple.

```
$ kill −SIGINT 3265 // interromp le processus de pid 3265}
$ kill −SIGSTOP 3255 // stoppe temporairement le processus de pid 3255}
$ kill −SIGCONT 4402 // reprend l'exécution du processus de pid 4402}
```

4.2 La fonction kill

Les processus ayant le même propriétaire peuvent communiquer entre eux en utilisant les signaux. La primitive « kill() » permet d'envoyer un signal « sig » vers un ou plusieurs processus.

Syntaxe int kill(pid_t pid, int sig) ;

« sig » désigne le signal à envoyer (on donne le nom ou le numéro du signal). Quand « sig » est égal à 0 aucun signal n'est envoyé, mais la valeur de retour de la primitive « kill() »

permet de tester l'existence ou non du processus « pid » (si kill(pid,0) retourne 0 (pas d'erreur) le processus de numéro « pid » existe).

- « pid » désigne le ou les processus destinataires (récepteurs du signal).
- Si pid > 0, le signal est envoyé au processus d'identité « pid ».
- Si pid=0, le signal est envoyé à tous les processus qui sont dans le même groupe que le processus émetteur (processus qui a appelé la primitive « kill() »).

Remarque : Cette possibilité peut être utilisée avec la commande shell « % kill -9 0 » pour tuer tous les processus en arrière-plan sans indiquer leurs identificateurs de processus).

- Si pid=-1, le signal est envoyé à tous les processus.
- Si pid < -1, le signal est envoyé à tous les processus du groupe de numéro |pid|.
- La primitive «kill()» renvoie 0 si le signal est envoyé et -1 en cas d'échec.

Exemple 4.1

```
void main(void) {
pid_t p;
if ((p=fork()) == 0) { /* processus fils qui boucle */
while (1); exit(2);
}    /* processus père */
sleep(10);
printf("Fin du processus père %d : %d\n", getpid());
}
```

Résultats d'exécution :

```
./test1 & ps -f
UID         PID         PPID   ... CMD
mak         2763        2761   ... bash
mak         3780        2763   .... ./test1
mak         3782        3780   ... ./test1
% Fin du processus père: 3780
Après quelques secondes, le programme affiche le message prévu dans «printf()» et se termine.
ps -f
UID         PID         PPID        ... CMD
mak         2763        2761        ... bash
mak         3782        1           ... ./test1
Remarque:
```

Le processus père a terminé son exécution mais le fils continue son exécution et devient orphelin (devient le fils du processus «init»).

Exemple 4.2

On modifie le programme précédent en envoyant un signal au processus fils. Ce signal provoquera la terminaison du processus fils.

```
void main(void) {
pid_t p;
if ((p=fork()) == 0) { /* processus fils qui boucle */
while (1);    exit(2);
}    /* processus père */
sleep(10);
printf("Envoi de SIGUSR1 au fils %d\n", p);
kill(p, SIGUSR1);
printf("Fin du processus père %d\n", getpid());    }
```

Résultats d'exécution :

```
./test1 & ps -f
UID          PID    PPID   ... CMD
mak          2763   2761   ... bash
mak          4031   2763   .... ./test1
mak          4033   4031   ... ./test1
% Envoi de SIGUSR1 au fils 4033
Fin du processus père 4031
Après quelques secondes, le programme affiche les message prévus dans «printf()» et se termine.
ps -f
UID    PID          PPID   ... CMD
mak    2763         2761   ... bash
```


Exemple 4.3: Envoie d'un signal à tous les processus

```
#include <signal.h>
void main(void) {
pid_t p1,p2;
if ((p1=fork()) == 0) { /* processus fils qui boucle */
        while (1);  exit(2);   }
else { if ((p2=fork()) == 0) { /* processus fils qui boucle */
        while (1);   exit(1);   }
}   /* processus père */
sleep(10);
printf(« Envoi de SIGUSR1 aux fils %d et %d \n", p1,p2);
kill(0, SIGUSR1);
printf("Fin du processus père %d\n", getpid());  }
```

Résultats d'exécution :

./test1 & ps -f

UID	*PID*	*PPID*	*... CMD*
mak	*1635*	*1633*	*... bash*
mak	*2007*	*1635*	*.... ./test1*
mak	*2008*	*2007*	*... ./test1*
mak	*2009*	*2007*	*... ./test1*

% Envoi de SIGUSR1 aux fils 2008 et 2009

Après quelques secondes, le programme affiche les message prévu dans « printf() » et se termine.

ps -f

UID	*PID*	*PPID*	*... CMD*
mak	*1635*	*1633*	*... bash*

Remarque: *Le processus père a interrompu l'exécution de ses deux fils (envoi du signal «SIGUSR1»).*

Exemple 4.4: Tester si un processus spécifié existe

```
void main(void) {
.....
sleep(2);
if ((kill(p, 0)==0)
printf(" le processus %d existe d\n",p);
else
printf(" le processus %d n'existe pas d\n",p );
}
```

4.3 Mise en place d'un handler

Les signaux, autres que « SIGKILL », «SIGCONT» et «SIGSTOP», peuvent avoir un handler spécifique installé par un processus. La primitive « **signal()** » peut être utilisée pour installer des handlers personnalisés pour le traitement des signaux. Cette primitive fait partie du standard de C.

Syntaxe: #include<signal.h> signal (int sig, new_handler);

Elle met en place le handler spécifié par « new_handler() » pour le signal « sig ». La fonction « new_handler » est exécutée par le processus à la réception du signal. Elle reçoit le numéro du signal. A la fin de l'exécution de cette fonction, l'exécution du processus reprend au point où elle a été suspendue.

Exemple 4.5

```
int main(void) {
for (;;) { }
return 0;
}
```

Résultat d'exécution: %./test

Le programme boucle.

Si on appuie sur CTRL-C le programme s'arrête

Exemple 4.6 : On met en place un handler qui permet de terminer le processus seulement lorsqu'on appuie deux fois sur «Ctr-C» (signal «SIGINT»).

```
void hand(int signum) {
printf(" Numéro du signal est %d \n", signum);
printf("Il faut appuyer sur Ctrl-C une 2ème fois pour terminer\n");
/* rétablir le handler par défaut du signal « SIGINT » en utilisant la macro « SIG_DFL » */
```

```
signal(SIGINT, SIG_DFL);  }
int main(void) {
signal(SIGINT, hand);       /* installation du nouveau handler */
for (;;) { } /* boucle infinie */ }
```

Exemple 4.7 On reprend l'exemple précédent exemple 4.6

```
void main(void) {
pid_t p;
if ((p=fork()) == 0) {/* processus fils qui boucle */
/* Installe le handler « SIG_IGN » pour le signal « SIGUSR1 » */
signal(SIGUSR1, SIG_IGN);
while (1);    exit(2);    }
      /* processus père */
sleep(10);
printf("Envoi de SIGUSR1 au fils %d\n", p);
kill(p, SIGUSR1);
printf("Fin du processus père %d\n", getpid());    }
```

Résultat d'exécution

Après quelques secondes, le programme affiche le message prévu dans «printf()» et se termine.

Le PPID du processus fils devient le processus « init » (PID=1) et ne se termine pas (boucle infini) car le signal « SIGUSR1 » a été ignoré.

5. sigaction( )

La gestion des signaux à la manière **Posix**.1(Portable Operating System Interface) n'est pas beaucoup plus compliquée que ce que nous avons vu dans le chapitre précédent. L'appel-système **sigaction( )** que nous allons étudier tout d'abord permet de réaliser toutes les opérations de configuration du gestionnaire et du comportement des signaux.

5.1 Structure « sigaction »

La primitive « **sigaction**() » peut être utilisée pour installer un handler personnalisé pour le traitement d'un signal. Elle prend comme arguments des pointeurs vers des structures « **sigaction** » définies comme suit: struct sigaction {

void (*sa_handler)() ;

sigset_t sa_mask ;

int sa_flags ; }

Le champ « **sa_handler** » pointeur vers une fonction (un handler) qui sera chargé de gérer le signal. La fonction peut être:

- « SIG_DFL » : correspond à l'action par défaut pour le signal.
- « SIG_IGN » : indique que le signal doit être ignoré.
- une fonction de type void qui sera exécutée par le processus à la réception du signal. Elle reçoit le numéro du signal.

Le champ « **sa_mask** » correspond à l'ensemble de signaux supplémentaires, à masquer (bloquer) pendant l'exécution du handler, en plus des signaux déjà masqués.

Le champ « **sa_flags** »: indique des options liées à la gestion du signal

Remarque:

Le champ «**sa_handler** » est obligatoire.

5.2 Primitive « sigaction() »

La fonction « sigaction() » permet d'installer un nouveau handler pour la gestion d'un signal.

Syntaxe: #include <signal.h>

int sigaction(int sig, struct sigaction *action, struct sigaction *action_ancien);

« sig »: désigne le numéro du signal pour lequel le nouveau handler est installé.

« action »: désigne un pointeur qui pointe sur la structure « sigaction » à utiliser. La prise en compte du signal entraine l'exécution du handler spécifié dans le champ « sa_handler » de la structure «action».

Si la fonction n'est ni « SIG_DFl » ni «SIG_IGN», le signal « sig » ainsi que ceux contenus dans la liste « sa_mask » de la structure « action » seront masqués pendant le traitement.

« action_ancien » contient l'ancien comportement du signal, on peut le positionner NULL.

Remarque:

Lorsqu'un handler est installé, il restera valide jusqu'à l'installation d'un nouveau handler.

Exemple:

Le programme suivant (« test_sigaction.c ») masque les signaux «SIGINT» et «SIGQUIT» pendant 30 secondes et ensuite rétablit les traitements par défaut.

```
struct sigaction action;
int main(void) {
action.sa_handler=SIG_IGN
sigaction(SIGINT,&action, NULL);
sigaction(SIGQUIT,&action, NULL);
printf("les signaux SIGINT et SIGQUIT sont ignorés \n");
sleep(30);
printf("Rétablissement des signaux \n");
action.sa_handler=SIG_DFL ;
sigaction(SIGINT,&action, NULL);
sigaction(SIGQUIT,&action, NULL);
while(1); }
```

Résultat d'exécution

```
./test_sigaction
les signaux SIGINT et SIGQUIT sont ignorés
// on appuie sur CTRL-C out CTRL –Q, il ne se //passe rien. Après 30 secondes on a l'affichage //suivant:
Rétablissement des signaux
// Maintenant, si on appuie sur CTRL-C ou CTRL-//Q, le programme s'arrête.
```

6. Les fonction « pause () » , « raise () » et « alarm() »

6.1 « pause () »

Un processus peut se mettre en veille et attendre l'arrivée d'un signal particulier pour exécuter le gestionnaire, ou bien l'émetteur utilise sleep()pour se synchroniser avec le récepteur.

pause() permet que le processus passe à l'état suspendu. Il va quitter cet état au moment de recevoir un signal. Après avoir reçu le signal, le processus recevra un traitement adéquat associé au signal et continuera son exécution à l'instruction qui suit pause(). Il retourne toujours la valeur -1. Cet appel fournit une alternative à l'attente active. C'est-à-dire qu'au lieu d'exécuter une boucle inutile en attente d'un signal, le processus bloquera jusqu'à ce qu'un signal lui soit transmis.

Syntaxe: #include<unistd.h>

int pause (void);

A la prise en compte d'un signal, le processus peut : se terminer car le « handler » associé est « SIG_DFL » ou exécuter le « handler » correspondant au signal intercepté.

Remarque: « pause() » ne permet pas d'attendre un signal de type donné ni de connaître le nom du signal qui l'a réveillé.

Exemple 6.1 : Attente d'un signale

```
int nb_req=0;    int pid_fils, pid_pere;
void Hand_Pere( int sig ){
nb_req ++;     printf("\t le pere traite la requête numero %d du fils \n", nb_req);
}
void main() {
if ((pid_fils=fork()) == 0) { /* FILS */
 pid_pere = getppid();
sleep(2);   /* laisser le temps au pere de se mettre en pause */
for (i=0; i<10; i++) {
  printf("le fils envoie un signal au pere\n");
kill (pid_pere, SIGUSR1);       /* demande de service */
sleep(2);       /* laisser le temps au père de se mettre en pause */
```

```
} exit(0);
}
else {       /* PERE */
signal(SIGUSER1,Hand_Pere);
while(1) {
pause();       /* attend une demande de service */
sleep(5); /* réalise le service */
 } } }
```

6.2 « raise () »

Syntaxe: int raise (int sig)

La fonction « int raise (int sig) » envoie le signal de numéro « sig » au processus courant (au processus qui appelle « raise() »). raise(sig) est équivalent à kill(getpid(),sig).

6.3 « alarm() »

L'appel système « **alarm()** » permet à un processus de gérer des temporisation(timeout) avant la routine concernée. Le processus est avertit de la fin d'une temporisation par un signal « **SIGLRM** ».

Syntaxe: #include<signal.h> **unsigned int alarm(unsigned int sec)**

Pour annuler une temporisation avant qu'elle n'arrive à son terme, on fait « **alarm(0)** » c'est-à-dire si la routine se termine normalement avant le délai maximal, on annule la temporisation avec **alarm(0).**

Le traitement par défaut du signal est la terminaison du processus. Pour modifier le comportement on installe un nouveau handler.

Exemple 6.2: Le programme « test_alarm.c » suivant lit deux entiers. Ensuite demande de lire le résultat qui le compare avec le produit.

Si le temps de réponse est plus grand que 3 secondes le programme se termine avec un message d'alarme, sinon il termine normalement.

```
main() {
int a, b, resultat;
printf("Entrer deux entiers ");
scanf("%d%d",&a,&b);
printf(" Donner le résultat de a*b : ");
 alarm(3);
scanf("%d",&resultat);
alarm(0); // annule la temporisation
if(resultat == (a*b))    printf(" Résultats juste \n ");
Else    printf(" Résultat faut \n ");
printf(" Fin normale du programme ");
}
```

Exemple 6.3 : on installe un handler

```
void hand(int signum) {
printf(" délais dépassé. Num SIGLRM = %d \n" , signum);  exit(1);  }
main() {
int a, b, resultat;
signal(SIGLRM,hand);
printf("Entrer deux entiers ");    scanf("%d%d",&a,&b);
printf(" Donner le résultat de a*b : ");
alarm(3);
scanf("%d",&resultat);
alarm(0); // annule la temporisation
if(resultat == (a*b))
 printf(" Résultats juste \n ");
Else  printf(" Résultat faut \n ");
```

```
printf(" Fin normale du programme "); }
```

Résultat d'exécution 2 (on met plus de temps pour saisir le résultat)

```
%.test_alarm
Entrer deux entier 1 5
Donner le résultat de  a*b
délais dépassé. Num SIGLARM = 14
```


Chap. III : Communication entre processus

Les processus coopèrent souvent pour traiter un même problème. Ces processus s'exécutent en parallèle sur un même ordinateur (monoprocesseur ou multiprocesseurs) ou bien sur des ordinateurs différents. Ils doivent alors s'échanger des informations (**communication interprocessus**). Il existe plusieurs moyens de communication interprocessus. Nous pouvons citer, entre autres, les variables communes, les fichiers communs, les signaux, les messages et les tubes de communication.

I. Synchronisation

1. Introduction

Après création d'un processus par l'appel système « fork() », le processus crée et son père s'exécutent de façon concurrente.

Lorsqu'un processus se termine, il envoie le signal « SIGCHLD » à son père et passe dans l'état zombi tant que son père n'a pas pris connaissance de sa terminaison, c'est à dire que la mémoire est libérée (le fils n'a plus son code ni ses données en mémoire) mais que le processus existe toujours dans la table des processus. Une fois le père lit le code retour du fils, à ce moment il termine et disparaît complètement du système.

Dans le cas où le père meurt avant la terminaison de son fils, alors le processus fils sera rattaché au processus « init » (l'ID du processus « init » est 1).

Afin de synchroniser le père avec ses fils (le processus père attende la terminaison de ses fils avant de se terminer), on utilise les fonctions de synchronisation « wait() » et « waitpid() » qui permettent au processus père de rester bloqué en attente de la réception d'un signal « SIGCHLD ».

2. Processus zombie

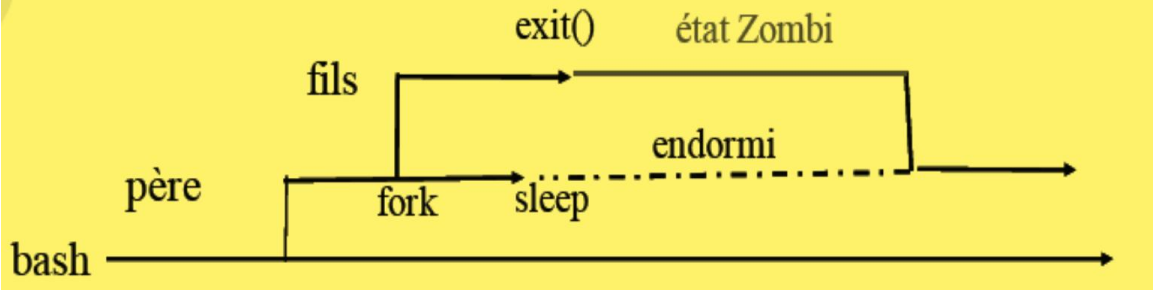
Au cours de leurs échanges avec le système et les programmes, les processus sont amenés à modifier leur état pour indiquer leur disponibilité. Ces changements sont le plus souvent dus à un besoin en ressources mémoire ou matérielle, à l'écriture de données ou encore à une attente (comme une action utilisateur).

Les états les plus connus sont l'état R (en cours d'exécution), S (en sommeil), T (stoppé) ou encore Z (zombie). Ce dernier est particulier, car il désigne un processus qui, bien qu'ayant terminé son exécution, reste présent sur le système, en attente d'être pris en compte par son père.

Quand un processus se termine (soit en sortant du « main() » soit par un appel à « exit() »), il délivre un code retour. Par exemple « exit(1) » donne un code retour égal à 1.

Tout processus qui se termine passe dans l'état « ***zombi »*** tant que son père n'a pas pris connaissance de sa terminaison.

Une fois le père ait connaissance de l'état de son fils, à ce moment le processus fils se termine et disparaît complètement du système



Le temps pendant lequel le processus fils est en état zombi, le fils n'a plus son code ni ses données en mémoire ; seules les informations utiles pour le père sont conservées.

Exemple 1 : Soit le fichier « test-zombi.c »

```
main() {
if (fork() == 0) { /* fils */
printf("je suis le fils, mon PID est %d\n", getpid());
sleep(1);
} else { /* père */
printf("je suis le père, mon PID est %d\n", getpid());
sleep(30);
} }
```

Dans ce cas le fils termine et attend que le père soit réveillé car le père est dans un état bloqué (sleep(30)) pour avoir connaissance de son état.

Pour que le fils se termine et disparait du système, il faut que le processus père lise l'état (le statut) de son fils.

```
./tes-zombi & ps g
je suis le fils, mon PID est 3748
je suis le père, mon PID est 3746
PID TTY STAT .... CMD
1716 pts/0 Ss .... bash
3746 pts/0 S ..... ./ test-zombi
3747 pts/0 R+ ..... ./ps g
3748 pts/0 S ..... ./test-zombi
```

```
% ps g
PID TTY STAT .... CMD
1716 pts/0 Ss .... bash
3746 pts/0 S ..... ./zombi
3748 pts/0 Z ..... [zombi] <defunct>
3749 pts/0 R+ ..... ps g
```

Les processus orphelins

La terminaison d'un processus parent ne termine pas ses processus fils. Dans ce cas les processus fils deviennent orphelins et sont adoptés par le processus initial (PID 1), c'est-à-dire que « init » devient leur père.

Exemple: Soit le fichier « test_orp.c »

```
main() {
if (fork() == 0) { /* fils */
printf("je suis le fils, mon PID est %d\n", getpid());
sleep(30); exit(0);
} else { /* père */
printf("je suis le père, mon PID est %d\n", getpid());
exit(0); } }
```

Après compilation on lance « test_orp » en background, ensuite en lance la commande « ps -f »

```
./test_orp & ps -f
Je suis le fils, mon PID est 3790
Je suis le père, mon PID est 3788
[2] 3788
UID PID        PPID   TTY .... CMD
```

```
mak 1969      1750    pts/0 .... bash
mak 3789      1969    pts/0 ..... ps -f
mak 3790      1       pts/0 ..... ./test_orp
```

3. Synchronisation père et fils : Les appels système wait(), waitpid()

Ces appels système permettent au processus père d'attendre la fin d'un de ses processus fils et de récupérer son status de fin. Ainsi, un processus peut synchroniser son exécution avec la fin de son processus fils en exécutant l'appel système wait().

La fonction **wait**() est déclarée dans <sys/wait.h>.

Syntaxe: # include<sys/wait.h> **pid_t wait (int * status);**

L'appel de la fonction **wait()** bloque le processus qui l'appelle en attente de la terminaison de l'un de ses processus fils.

- Si le processus qui appelle **wait()** n'a pas de fils, alors **wait()** retourne -1.
- Sinon **wait()** retourne le PID du processus fils qui vient de se terminer.

Si le processus appelant admet au moins un fils en attente à l'état zombie, alors **wait()** revient immédiatement et renvoie le PID de l'un de ces fils zombis.

« status » est un pointeur sur un « int ». Si le pointeur « status » est non NULL, il contient des informations sur la terminaison du processus fils (code du signal qui a tué le fils, les circonstances de la terminaison du fils).

Pour la lecture et l'analyse des code retour, on a les macros suivantes :

- « WEXITSTATUS(status) »: fournit le code retour du processus s'il s'est terminé normalement.
- « WIFEXITED(status) »: vrais si le processus fils s'est terminé normalement (en sortant du « main() » ou par un un appel à « exit() »).
- « WIFSIGNALED(status) »: vrai si le fils s'est terminé à cause d'un signal.
- « WTERMSIG(status) »: fournit le numéro du signal ayant tué le processus.
- « WIFSTOPPED(status) »: indique que le processus fils est stoppé temporairement.
- « WSTOPSIG(status) »: fournit le numéro du signal ayant stoppé le processus.

Remarque : Pour attendre la terminaison de tous les fils il faut faire une boucle sur le nombre de fils. Supposons que le processus qui appelle «wait()» a « nb_fils » alors il appelle la fonction « wait() » « nb_fils » fois .

```
for(i=1; i<=nb_fils; i++)
wait(0); /* ou wait(NULL); */
ou même
while (wait(0)!=-1); /* while (wait(NULL)!=-1); */
```

Il y a deux inconvénients avec la fonction wait( ) , qui ont conduit à développer la fonction waitpid( ) que nous allons voir ci-dessous. Le premier problème, c'est que l'appel reste bloquant tant qu'aucun fils ne s'est terminé. Il n'est donc pas possible d'appeler systématiquement wait( ) dans une boucle principale du programme pour savoir où en est le fils. La solution est d'installer un gestionnaire pour le signal SIGCHLD qui est émis dès qu'un fils se termine ou est stoppé temporairement.

Le second problème vient du fait qu'il n'est pas possible d'attendre la fin d'un fils particulier. Dans ce cas, il faut alimenter dans le gestionnaire du signal SIGCHLD une liste des fils terminés, qu' on consultera dans le programme principal en attente d'un processus donné. Il ne faut pas oublier de bloquer temporairement SIGCHLD lors de la consultation de la liste, pour éviter qu'elle ne soit modifiée pendant ce temps par l'arrivée d'un signal.

Pour pallier ces deux problèmes, un appel-système supplémentaire est disponible, waitpid( ), dont le prototype est déclaré dans <sys /wait.h> ainsi : La fonction « waitpid() » permet d'attendre un processus fils particulier ou appartenant à un groupe.

Syntaxe: pid_t waitpid (pid_t pid, int * status, int options);

- « pid »: désigne le pid du processus fils qu'on attend sa terminaison.
  - Si pid >0: processus père attend le processus fils identifié par «pid».
  - Si pid=0, le processus appelant attend la terminaison de n'importe quel processus fils appartenant au même groupe que le processus appelant.
  - Si pid =-1: le processus père attend la terminaison de n'importe quel processus fils, comme avec la fonction « wait() ».
  - Si pid<-1: le processus père attend la terminaison de n'importe quel processus fils appartenant au groupe de processus dont le numéro est |pid|.
- « status »: a exactement le même rôle que la fonction «wait()».

- « options »: permet de préciser le comportement de «waitpid()», les constantes suivantes :
  - WNOHANG: le processus appelant n'est pas bloqué si le processus spécifié n'est pas terminé. Dans ce cas, waitpid() renverra 0.
  - WUNTRACED: si le processus spécifié est stoppé, on peut accéder aux informations concernant les processus fils temporairement stoppés en utilisant les macros WIFSTOPPED(status) et WSTOPSIG(status) .

L'appel le plus simple est: pid_t waitpid (pid_t pid, int * status, 0); En cas de succès, elle retourne le « pid » du processus fils attendu.

Exemple 3.1 :

```
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
 int  main(int argc, char *argv[])
     {
        pid_t cpid, w;
        int status;

        cpid = fork();
        if (cpid == -1) {
           perror("fork");
           exit(EXIT_FAILURE);
        }
        if (cpid == 0) {          /* Code exécuté par le fils */
           printf("Le PID du fils est %ld\n", (long) getpid());
           if (argc == 1)
              pause();                  /* Attendre un signal */
           _exit(atoi(argv[1]));

        } else {                  /* Code exécuté par le père */
           do {
              w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);
              if (w == -1) {
                 perror("waitpid");
                 exit(EXIT_FAILURE);
              }
              if (WIFEXITED(status)) {
                 printf("terminé, code=%d\n", WEXITSTATUS(status));
              } else if (WIFSIGNALED(status)) {
```

```
            printf("tué par le signal %d\n", WTERMSIG(status));
          } else if (WIFSTOPPED(status)) {
            printf("arrêté par le signal %d\n", WSTOPSIG(status));
          } else if (WIFCONTINUED(status)) {
            printf("relancé\n");
          }
        } while (!WIFEXITED(status) && !WIFSIGNALED(status));
        exit(EXIT_SUCCESS);
      }
    }
```

Exemple d'exécution :

```
./a.out &
        Le PID du fils est 3360
        [1] 3359
        $ kill -STOP 3360
        arrêté par le signal 19
        $ kill -CONT 3360
        relancé
        $ kill -TERM 3360
        tué par le signal 15
        [1]+  Done                    ./a.out
```

4. Opérations sur les ensembles de signaux

Le type « **sigset_t** » désigne un ensemble de signaux. On peut manipuler les ensemble de signaux à l'aide des fonctions suivantes :

int sigemptyset (sigset_t * ens); Initialise l'ensemble de signaux « ens » à l'ensemble vide.

int sigfillset (sigset_t * ens); Remplit l'ensemble « ens » avec tous les signaux.

int sigaddset (sigset_t * ens, int sig); Ajoute le signal « sig » à l'ensemble « ens ».

int sigdelset (sigset_t * ens, int sig); Retire le signal « sig » de l'ensemble « ens ».

int sigismember (const sigset_t * ens, int sig); Retourne vrai si le signal « sig » appartient à l'ensemble « ens ».

Blocage ou Masquage des signaux

ensemble de signaux, sauf SIGKILL et SIGSTOP. Cette opération se fait principalement grâce à l'appel-système sigprocmask( ). Cette routine est très complète puisqu'elle permet aussi bien de bloquer ou de débloquer des signaux, que de fixer un nouveau masque complet ou de consulter l'ancien masque de blocage.

La primitie « **sigprocmask()** » permet de masquer ou de démasquer un ensemble de signaux. La valeur de retour de la fonction est 0 ou -1 selon qu'elle est ou non bien déroulée

Syntaxe: #include <signal.h>

int sigprocmask(int opt, const sigset_t *ens, sigset_t *ens_ancien);

- « ens » pointeur sur le nouveau ensemble des signaux à masquer.
- « ens_ancien »: pointeur sur le masque antérieur. Il est récupéré au retour de la fonction.
- « opt » précise ce que on fait avec ces ensembles. Les valeurs de « opt » sont des constantes symboliques définies dans « signal.h ».

Si le troisième argument « ens_ancien » est un pointeur non NULL, alors la fonction installe un masque à partir des ensembles pointés par « ens » et « ens_ancien ».

Valeur du paramètre opt :

- « SIG_SETMASK »

  Les seuls signaux masqués seront ceux de l'ensemble «ens».

  Nouveau masque = « ens ».
- « SIG_BLOCK »:

  Les signaux masqués seront ceux des ensembles « ens » et «ens_ancien».

  Nouveau masque = « ens » U « ens_ancien »
- « SIG_UNBLOCK »

  Démasquage des signaux de l'ensemble « ens ». Nouveau masque :

  «ens_ancien» - «ens »

Liste des signaux pendant

Il est important qu'un processus puisse consulter la liste des signaux bloqués en attente, sans pour autant en demander la délivrance immédiate. Cela s'effectue à l'aide de l'appel système sigpending( ),

Grâce à la fonction « sigpending() » on peut voir la liste des signaux, en attente d'être pris en compte. Le code retour de la fonction est 0 si elle est bien déroulée ou -1 sinon.

Syntaxe: #include <signal.h> int sigpending(sigset_t *ens) ;

Retourne dans « ens » la liste des signaux bloqués (en attente d'être pris en compte).

Exemple 4.1 : Le programme suivant (**test_pending.c**) montre le masquage et le démasquage d'un ensemble de signaux et comment connaître les signaux pendants.

```
sigset_t ens1, ens2;
int sig;
main() {
//construction de l'ensemble ens1={SIGINT, SIGQUIT, SIGUSR1}
sigemptyset(&ens1);
sigaddset(&ens1,SIGINT); // ajoute le signal SIGINT à ens1
sigaddset(&ens1,SIGQUIT); // ajoute le signal SIGQUIT à ens1
sigaddset(&ens1,SIGUSR1); // ajoute le signal SIGUSR1 à ens1
printf(" Numéros des signaux %d %d %d\n", SIGINT, SIGUSR1 ,SIGQUIT );
// Installation du masque ens1
sigprocmask(SIG_SETMASK, &ens1, NULL);
printf("Masquage mis en place.\n");
sleep(15);
/* affichage des signaux pendants: signaux envoyés mais non délivrés car masqués*/
sigpending(&ens2);
printf("Signaux pendants:\n");
for(sig=1; sig<NSIG; sig++)
if(sigismember(&ens2, sig))    printf("%d \n",sig);
sleep(15); /* Suppression du masquage des signaux */
sigemptyset(&ens1);
```

```
printf("Déblocage des signaux.\n");
sigprocmask(SIG_SETMASK, &ens1, (sigset_t *)0); NULL
sleep(15);
printf("Fin normale du processus \n");
exit(0); }
```

Résultat d'exécution 1er test: on lance l'exécution et on attend la fin du programme

```
./test_pending
Numéro des signaux 2 10 3
Masquage mis en place
Signaux pendant:
Déblocage des signaux
Fin normal du processus
```

Résultat d'exécution 2ème test: on lance l'exécution et ensuite on tappe Ctrl-C

```
./test_pending
Numéro des signaux 2 10 3
Masquage mis en place
^C Signaux pendant:
2
Déblocage des signaux
```

Résultat d'exécution : 3ème test : on lance l'exécution et ensuite on tappe Ctrl-C et Ctrl-Quit

```
./test_pending
Numéro des signaux 2 10 3
Masquage mis en place
^C ^\ Signaux pendant :
23
Déblocage des signaux
```

II. Communication par les tubes (pipes)