Unix : Signaux

Auteur: Zouheir HAMROUNI

2 Les Signaux

2.1 Introduction

Les signaux constituent un mécanisme fondamental de communication entre processus. Un signal représente un événement externe au processus, provoqué par une erreur (accès mémoire invalide par exemple), ou envoyé expilicitement par un autre processus.

Il en existe une trentaine, identifiés par leur numéro ou leur nom, dont voici une sous-liste (nom, numéro, rôle (version / norme)). Les noms sont standards mais les numéros peuvent changer selon le système :

```
SIGINT          2       Interrupt                       (ANSI)
SIGQUIT         3       Quit                            (POSIX)
SIGILL          4       Illegal instruction             (ANSI)
SIGTRAP         5       Trace trap                      (POSIX)
SIGBUS          7       BUS error                       (4.2 BSD)
SIGFPE          8       Floating-point exception        (ANSI)
SIGKILL         9       Kill, unblockable               (POSIX)
SIGUSR1         10      User-defined signal 1           (POSIX)
SIGSEGV         11      Segmentation violation          (ANSI)
SIGUSR2         12      User-defined signal 2           (POSIX)
SIGPIPE         13      Broken pipe                     (POSIX)
SIGALRM         14      Alarm clock                     (POSIX)
SIGTERM         15      Termination                     (ANSI)
SIGSTKFLT       16      Stack fault
SIGCHLD         17      Child status has changed        (POSIX)
SIGCLD          SIGCHLD Same as SIGCHLD                 (System V)
SIGCONT         18      Continue                        (POSIX)
SIGSTOP         19      Stop, unblockable               (POSIX)
SIGTSTP         20      Keyboard stop                   (POSIX)
SIGURG          23      Urgent condition on socket      (4.2 BSD)
SIGSYS          31      Bad system call
```

L'arrivée d'un signal est asynchrone, comme la sonnerie du téléphone : on sait que cela peut arriver, mais on ne décide pas de l'instant où cela arrive.

À chaque signal est associé un traitement, nommé "handler" ou traitant (sous-programme), qui est exécuté lorsque le signal est reçu par le processus. Chaque signal a un traitant par défaut, mais qui peut être modifié (sauf pour certains signaux). Il existe cinq types de traitements par défaut :

- la réception du signal arrête le processus (SIGINT, SIGQUIT, SIGKILL, ...)
- la réception du signal arrête le processus avec création d'un fichier core (SIGSEGV)
- le signal est ignoré (SIGCHLD signal délivré automatiquement au père lors de changement d'état de tout processus fils)
- la réception du signal suspend le processus (SIGSTOP, SIGTSTP)
- la réception du signal relance le processus suspendu (SIGCONT)

L'émission d'un signal peut être provoquée par :

- le système (erreur d'exécution, ou interruption système)
- une action de l'utilisateur depuis le terminal (CtrlC (SIGINT), CtrlZ (SIGTSTP))
- un signal envoyé par un autre processus :
  - primitive `kill(num_sig, pid)`
  - depuis un autre terminal : kill num_signal pid

Première illustration

1. Dans un premier terminal (T1), exécuter la commande `sleep 300`
2. Dans un second terminal (T2), exécuter la commande `ps l`, et noter le pid du processus "sleep"
3. Dans le premier terminal (T1), taper en même temps les touches `Ctrl` et `Z`. Par la suite, on notera Ctrl+Z;
4. Dans le second terminal (T2), exécuter la commande `ps l`, et noter l'état du processus "sleep" (Il est suspendu)
5. Dans le second terminal (T2), exécuter la commande `kill -CONT pid_du_sleep_300`, et vérifier son état avec la commande `ps l`. On remarquera que le prcessus "sleep" a été relancé, mais qu'il ne reprend la main sur le terminal T1. Il continue à s'exécuter en arrière plan. On verra dessous comment le ramener en avant plan.
6. Dans le premier terminal (T1), exécuter la commande `sleep 200`
7. Dans le second terminal (T2), exécuter la commande `kill -TSTP pid_du_sleep_200`, et vérifier ce qu'il se passe dans le terminal T1
8. Dans le premier terminal (T1), exécuter la commande `jobs` qui affiche la table des porcessus en arrière plan
9. Dans le premier terminal (T1), exécuter la commande `fg` avec argument le numéro du processus sleep_300 dans la table affichée dessus : on remarquera que la commande "fg" (foregroung) a ramené le porcessus en avant plan.
10. Dans ce terminal (T1), taper les touches Ctrl+C, et observer ce qu'il se passe

2.2 Nouveau traitant

Il est possible de modifier le traitement associé un signal donné (sauf pour certains) :

1. en créant un traitant (handler) sous la forme d'une procédure. Ce traitant ne peut avoir qu'un seul paramètre : le numéro du signal.

```
void handler (int num_sig)
{
   printf("signal recu : %d\n", num_sig);
   ...
}
```

2. et en associant ce traitant au signal souhaité, avec la primitive `signal` qui prend deux paramètres : le numéro du signal et la procédure à exécuter lorsque ce signal est reçu. La fonction `signal` renvoie la référence du traitant qui était précédemment associée au signal.

```
...
signal(sig, handler);
...
```

Lorsque le signal est reçu :

- l'exécution du code courant est interrompue
- le numéro du signal est alors fourni comme argument au handler associé
- le handler est exécuté
- si le handler ne provoque un arrêt, l'exécution du processus reprend au point où il avait été interrompu

Illustration

On reprend le code initial "pere_fils.c" et on y apporte les ajouts suivants :

- on définit un traitant (handler_sigint) qui se contente d'afficher le pid du processus et le numéro du signal qu'il a reçu

- on associe ce traitant au signal SIGINT
- on définit une procédure dormir qui permet au processus appelant de s'endormir pendant nb_secondes, valeur passée en paramètre. L'appel à la fonction sleep sera remplacé parun appel à dormir (qui exécute des appels succesifs à sleep(1)), car la réception d'un signal peut engendrer une sortie du sleep.

Ce qui donne le code suivant :

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* Traitement du signal SIGINT */

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */
#include <signal.h>   /* traitement des signaux */

#define NB_FILS 3     /* nombre de fils */

/* Traitant du signal SIGINT */
void handler_sigint(int signal_num) {
    printf("\n     Processus de pid %d : J'ai reçu le signal %d\n",
            getpid(),signal_num) ;
    return ;
}

/* dormir pendant nb_secondes secondes  */
/* à utiliser à la palce de sleep(duree), car sleep s'arrête */
/* dès qu'un signal non ignoré est reçu */
void dormir(int nb_secondes)
{
    int duree = 0 ;
    while (duree < nb_secondes) {
        sleep(1) ;
        duree++ ;
    }
}

int main()
{
    int fils, retour ;
    int duree_sommeil = 600 ;

    /* associer un traitant au signal SIGINT */
    signal(SIGINT, handler_sigint) ;

    printf("\nJe suis le processus principal de pid %d\n", getpid()) ;

    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ;

        /* Bonne pratique : tester systématiquement le retour des appels système */
        if (retour < 0) {   /* échec du fork */
            printf("Erreur fork\n") ;
            /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
            exit(1) ;
        }

        /* fils */
        if (retour == 0) {
            printf("\n     Processus fils numero %d, de pid %d, de pere %d.\n",
                    fils, getpid(), getppid()) ;
            dormir(duree_sommeil) ;
            /* Important : terminer un processus par exit */
            exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
        }

        /* pere */
        else {
            printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
                    getpid(), fils, retour) ;
        }
    }
```

```
65     dormir(duree_sommeil + 2) ;
66     return EXIT_SUCCESS ;
67 }
```

On peut tester le fonctionnement de ce programme dans l'outil "apisys", ou en local selon l'enchaînement suivant :

1. lancer son exécution dans un terminal (T1)
2. dans un second terminal (T2), taper la commande kill -INT pid_du_fils_numéro_1
3. observer ce qu'il se passe dans le premier terminal (T1), et noter le numéro du signal SIGINT
4. dans ce premier terminal (T1), taper Ctrl+C et observer ce qu'il se passe et noter le numéro du signal provoqué par CtrlC. Il y a une petite différence dans le nommage des signaux en C et en shell : SIGINT en C, mais INT en shell.

On peut remarquer que :

- l'association entre le signal SIGINT et handler_sigint a été effectuée dans le code du père avant la création des 3 fils
- que ces 3 fils ont hérité de ce traitement : ils ont eu le même comportement que leur père vis à vis du signal SIGINT

Travail complémentaire

Transformer ce code en assoicant le même traitant pour les signaux SIGINT, SIGTSTP et SIGCONT.

Exécuter et noter les numéros des signaux SIGTSTP et SIGCONT, ainsi que le signal provoqué par Ctrl+Z.

2.3 SIG_IGN et SIG_DFL

Comme indiqué plus haut, il existe un traitement par défaut pour chacun des signaux. Et on peut donc revenir au traitant par défaut, référencé par SIG_DFL, en l'associant de nouveau au signal souhaité. Par exemple :

```
signal(SIGINT, SIG_DFL)
```

On peut aussi ignorer un signal (sauf certains comme le signal KILL), en associant le traitant référencé par SIG_IGN au signal souhaité. Par exemple :

```
signal(SIGINT, SIG_IGN)
```

Pour illustrer cela, on reprend le code précédent et on y apporte les deux modifications suivantes :

- Pour le fils 1 : on ignore le signal SIGINT (signal(SIGINT, SIG_IGN);)
- Pour le fils 2 : on associe le traitant par défaut à SIGINT (signal(SIGINT, SIG_DFL);)
- Le fils 3 garde le même traitant que son père

Ce qui donne le code suivant :

```
1 /* Exemple d'illustration des primitives Unix : Un père et ses fils */
2 /* Traitement du signal SIGINT : SIG_IGN et SIG_DFL */
3
4 #include <stdio.h>    /* entrées sorties */
5 #include <unistd.h>   /* pimitives de base : fork, ...*/
6 #include <stdlib.h>   /* exit */
7 #include <signal.h>   /* traitement des signaux */
8
9 #define NB_FILS 3     /* nombre de fils */
10
11 /* Traitant du signal SIGINT */
```

```c
void handler_sigint(int signal_num) {
    printf("\n     Processus de pid %d : J'ai reçu le signal %d\n",
            getpid(),signal_num) ;
    return ;
}

/* dormir pendant nb_secondes secondes  */
/* à utiliser à la palce de sleep(duree), car sleep s'arrête */
/* dès qu'un signal non ignoré est reçu */
void dormir(int nb_secondes)
{
    int duree = 0 ;
    while (duree < nb_secondes) {
        sleep(1) ;
        duree++ ;
    }
}

int main()
{
    int fils, retour ;
    int duree_sommeil = 600 ;

    /* associer un traitant au signal SIGINT */
    signal(SIGINT, handler_sigint) ;

    printf("\nJe suis le processus principal de pid %d\n", getpid()) ;

    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ;

        /* Bonne pratique : tester systématiquement le retour des appels système */
        if (retour < 0) {   /* échec du fork */
            printf("Erreur fork\n") ;
            /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
            exit(1) ;
        }

        /* fils */
        if (retour == 0) {
            if (fils == 1) {
                signal(SIGINT, SIG_IGN) ;
            }
            else if (fils == 2) {
                signal(SIGINT, SIG_DFL) ;
            }
            printf("\n     Processus fils numero %d, de pid %d, de pere %d.\n",
                    fils, getpid(), getppid()) ;
            dormir(duree_sommeil) ;
            /* Important : terminer un processus par exit */
            exit(EXIT_SUCCESS) ;   /* Terminaison normale (0 = sans erreur) */
        }

        /* pere */
        else {
            printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
                    getpid(), fils, retour) ;
        }
    }
    dormir(duree_sommeil+2) ;
    return EXIT_SUCCESS ;
}
```

On peut tester le fonctionnement de ce programme dans l'outil "apisys", ou en local selon l'enchaînement suivant :

1. lancer son exécution dans un premier terminal T1
2. dans un second terminal T2, taper la commande kill -INT pid_fils_1, et on remarquera que rien ne se passe (signal ignoré)
3. dans le second terminal T2, taper la commande kill -INT pid_fils_2, et on vérifiera avec la commande "ps l" que le fils 2 est tué (comportement par défaut)

4. dans le second terminal T2, taper la commande kill -INT pid_fils_3, et on remarquera qu'un message apparaît dans le terminal T1, indiquant la réception de ce signal par le fils 3 (comportement hérité du père)

2.4 exec et signaux

L'appel à la primite `exec` engendre deux comportement différents par rapports au traitement des signaux :

- le traitement SIG_IGN et SIG_DFL sont conservés
- un traitant particulier défini par l'appelant n'est pas conservé lors d'un appel à `exec`. Dans ce cas, c'est le traitant par défaut (SIG_DFL) aui est repris.

Pour illustrer cela, on reprend le code précédent, et on remplace les appels à la procédure "dormir", dans les trois fils, par le lancement avec execl de l'exécutable "dormir" fourni dans le tutoriel "processus". Ce qui donne le code suivant :

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* Traitement du signal SIGINT : SIG_IGN et SIG_DFL avec exec */

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */
#include <signal.h>   /* traitement des signaux */

#define NB_FILS 3     /* nombre de fils */

/* Traitant du signal SIGINT */
void handler_sigint(int signal_num) {
    printf("\n     Processus de pid %d : J'ai reçu le signal %d\n",
            getpid(),signal_num) ;
    return ;
}

/* dormir pendant nb_secondes secondes  */
/* à utiliser à la palce de sleep(duree), car sleep s'arrête */
/* dès qu'un signal non ignoré est reçu */
void dormir(int nb_secondes)
{
    int duree = 0 ;
    while (duree < nb_secondes) {
        sleep(1) ;
        duree++ ;
    }
}

int main()
{
    int fils, retour ;
    int duree_sommeil = 600 ;

    char ref_exec[]="./dormir" ; /* exécutable  */
    char arg0_exec[]="je dors" ;   /* argument0 du exec : nom donnée au processus */
    char arg1_exec[]="600" ;       /* argument0 du exec : durée de sommeil */

    /* associer un traitant au signal SIGINT */
    signal(SIGINT, handler_sigint) ;

    printf("\nJe suis le processus principal de pid %d\n", getpid()) ;

    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ;

        /* Bonne pratique : tester systématiquement le retour des appels système */
        if (retour < 0) {   /* échec du fork */
            printf("Erreur fork\n") ;
            /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
            exit(1) ;
        }
```

```
53
54          /* fils */
55          if (retour == 0) {
56              if (fils == 1) {
57                  signal(SIGINT, SIG_IGN) ;
58              }
59              else if (fils == 2) {
60                  signal(SIGINT, SIG_DFL) ;
61              }
62              printf("\n     Processus fils numero %d, de pid %d, de pere %d.\n",
63                      fils, getpid(), getppid()) ;
64              execl(ref_exec, arg0_exec, arg1_exec, NULL) ;
65              /* on ne se retrouve ici que si exec échoue */
66              printf("\n     Processus fils numero %d : ERREUR EXEC\n", fils) ;
67              /* perror : affiche un message relatif à l'erreur du dernier appel systàme */
68              perror("     exec ") ;
69              exit(fils) ;   /* sortie avec le numéro di fils qui a échoué */
70          }
71
72          /* pere */
73          else {
74              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
75                      getpid(), fils, retour) ;
76          }
77      }
78      dormir(duree_sommeil+2) ;
79      return EXIT_SUCCESS ;
80  }
```

On peut tester le fonctionnement de ce programme dans l'outil "apisys", ou en local selon l'enchaînement suivant :

1. lancer son exécution dans un terminal T1
2. dans un second terminal T2, taper la commande kill -INT pid_fils_1, et on remarquera que rien ne se passe (signal ignoré), donc traitement conservé
3. dans le second terminal T2, taper la commande kill -INT pid_fils_2, et on remarquera que le fils 2 est tué (commande ps l), donc traitement conservé
4. dans le second terminal T2, taper la commande kill -INT pid_fils_3, et on remarquera le fils 3 est tué (commande ps l), donc le traitant hérité du père a été remplacé par SIG_DFL

2.5 SIGCHLD

Lorsqu'un fils s'arrête, est suspendu ou relancé, le signal SIGCHLD est envoyé à son père. Par défaut, ce signal est ignoré, mais on peut choisir de lui associer un traitant en fonction des objectifs de l'application.

Si on reprend le programme pere_fils_wait.c, étudié dans le tutoriel "processus", on peut remarquer :

- que l'on a utilsé une boucle pour attendre la terminaison des 3 fils
- que durant cette attente, notre programme ne faisait rien d'autre, car cette attente était bloquante

Pour remédier à cela, on peut :

- décharger le programme principal de cette attente bloquante
- utiliser le signal SIGCHLD pour être notifié de la terminaison d'un fils, et lui associer un traitant qui effectue le traitement souhaité, par exemple compter le nombre de fils terminés, ou tenir à jour une liste des processus toujours en vie

On définit donc un traitant pour le signal SIGCHLD, dans lequel on effectue un appel à la primitive wait pour récupérer les informations conernant le fils venant de s'arrêter et on incrémente le nombre de fils terminés. Il faut noter que l'on ne restera pas bloqué sur le wait, car le retour sera immédiat puisqu'un fils vient de s'arrêter (raison de l'entrée dans la traitant).

Ce qui donne le code suivant :

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* utilisation de SIGCHLD pour prendre connaissance de la fin des fils */

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */
#include <sys/wait.h> /* wait */
#include <signal.h>   /* opérations sur les signaux */

#define NB_FILS 3     /* nombre de fils */

int nb_fils_termines ; /* variable globale car modifiée par le traitant */

/* Traitant du signal SIGCHLD */
void handler_sigchld(int signal_num) {
    int wstatus, fils_termine ;

    fils_termine = wait(&wstatus) ;
    nb_fils_termines++ ;
    if WIFEXITED(wstatus) {   /* fils terminé avec exit */
        printf("\nMon fils de pid %d a termine avec exit %d\n",
                fils_termine, WEXITSTATUS(wstatus)) ;
    }
    else if WIFSIGNALED(wstatus) {  /* fils tué par un signal */
        printf("\nMon fils de pid %d a ete tue par le signal %d\n",
                fils_termine, WTERMSIG(wstatus)) ;
    }
    return ;
}

int main()
{
    int fils, retour ;
    int duree_sommeil = 300 ;

    nb_fils_termines = 0 ;
    /* associer traitant sigchld à SIGCHLD */
    signal(SIGCHLD, handler_sigchld) ;

    printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
    /* Vidange du tampon de sortie pour que le fils le récupère vide        */
    fflush(stdout) ;

    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ;

        /* Bonne pratique : tester systématiquement le retour des appels système */
        if (retour < 0) {   /* échec du fork */
            printf("Erreur fork\n") ;
            /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
            exit(1) ;
        }

        /* fils */
        if (retour == 0) {
            printf("\n     Processus fils numero %d, de pid %d, de pere %d.\n",
                    fils, getpid(), getppid()) ;
            sleep(2 + duree_sommeil * (fils-1)) ;
            exit(fils) ;  /* normalement exit(0), mais on veut illustrer WEXITSTATUS */
        }

        /* pere */
        else {
            printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
                    getpid(), fils, retour) ;
        }
    }

    /* faire ce qu'on veut jusqu'à la terminaison de tous les fils */
    do {
        /* faire ce qu'on veut */
```

```
74
75      } while (nb_fils_termines < NB_FILS) ;
76      printf("\nProcessus Principal termine\n") ;
77      return EXIT_SUCCESS ;
78  }
```

On peut tester le fonctionnement de ce programme dans l'outil "apisys", ou en local selon l'enchaînement suivant :

1. lancer son exécution dans un terminal T1, et noter l'arrêt rapide du fils 1 : arrêt normal avec exit
2. dans un second terminal T2, taper la commande kill -QUIT pid_fils_2, et on verra s'afficher dans le terminal T1 un message indiquant que le père a pris connaissance de l'arrêt de son fils par le signal QUIT (3)
3. dans le second terminal T2, taper la commande kill -KILL pid_fils_3, et on verra s'afficher dans le terminal T1 un message indiquant que le père a pris connaissance de l'arrêt de son fils par le signal KILL (9)

Mais cette solution possède un inconvénient majeur :

- si un un signal arrive alors que l'on se trouve dans son traitant, il est mémorisé et sera traité après la sortie du traitant
- mais si un second arrive alors que l'on se trouve touours dans le traitant, il est ignoré, et sera définitivement perdu

On peut voir cette situation en mettant la même durée de sommeil pour les trois fils. Ce qui peut aboutir à la perte de l'un des signaux SIGCHLD. Deux ou trois exécutions pourraient être nécessaires pour mettre en évidence cette situation.

2.6 SIGCHLD et waitpid

Pour remédier à l'inconvénient de la solution précédente, une solution possible consiste à utiliser une boucle dans le handler pour prendre en compte la terminaison de tous fils qui ont eu lieu alors que l'on se trouve déjà dans le handler. Mais la primitive `wait` ne peut pas être utilisée dans cette boucle, car elle est bloquante.

On aura donc recours à une seconde primitive plus riche `pid_t waitpid(pid_t pid, int *status, int options)`, qui offre deux avantages par rapport à wait :

- elle peut être utilisée de façon non bloquante : teste si un fils s'est terminé et retourne immédiatement
- elle peut prendre en compte deux autres événements sur le fils : suspension et relance

Elle prend les paramètres suivants :

- pid : pid du fils que l'on souhaite attendre (-1 : n'importe leqel des processus fils)
- status (par référence) : contiendra au retour la cause de terminaision et la valeur correspondante. En plus des 4 macros utilisées dans le tutoriel "processus", on peut utiliser :
  - WIFSTOPPED(status) renvoie vrai si le fils a été suspendu par un signal
  - WSTOPSIG(status) renvoie le numéro du signal qui a causé la suspension du fils
  - WIFCONTINUED(status) renvoie vrai si le processus fils a été relancé par le signal SIGCONT.
- options : peut être une combinaison (OU binaire) entre différentes constances :
  - WNOHANG : retour immédiat même si aucun fils n'a pas changé d'état (non bloquant)
  - WUNTRACED : retour si un fils est suspendu, nécessaire pour WIFSTOPPED et WSTOPSIG
  - WCONTINUED : retour si un fils suspendu a été relancé par un signal SIGCONT, nécessaire pour WIFCONTINUED

On reprend le code précédent, et on apporte les modifications suivantes :

- on remplace l'appel à wait par un appel à waitpid en mode non bloquant, et on ajoute la prise en compte des événements de suspension et de relance
- on encapsule l'appel à waitpid dans une boucle while de manière à pouvoir capter tous les événements subis par l'un des fils alors que l'on se trouve déjà dans le traitant. De cette manière, même si un signal SIGCHLD est perdu, on traite quand même l'événement corresponsant dans la bucle while (retour > 0 de waitpid).
- Le père crée 4 fils : les 3 premiers se terminent après 2 secondes, et le dernier dure assez longtemps pour pouvoir recevoir différents signaux.

Cela donne le code suivant :

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* SIGCHLD et waitpid pour prendre connaissance de la fin des fils  */

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */
#include <sys/wait.h> /* wait */
#include <signal.h>   /* opérations sur les signaux */
#include <string.h>

#define NB_FILS 4     /* nombre de fils */

int nb_fils_termines ; /* variable globale car modifiée par le traitant */

/* Traitant du signal SIGCHLD */
void handler_chld(int signal_num) {
   int fils_termine, wstatus ;
   printf("\nJ'ai reçu le signal %d\n",  signal_num) ;
   if (signal_num == SIGCHLD) {
      while ((fils_termine = (int) waitpid(-1, &wstatus, WNOHANG | WUNTRACED | WCONTINUED)) > 0) {
         if WIFEXITED(wstatus) {
            printf("\nMon fils de pid %d s'est arrete avec exit %d\n",  fils_termine, WEXITSTATUS(wstatu
            nb_fils_termines++ ;
         }
         else if WIFSIGNALED(wstatus) {
            printf("\nMon fils de pid %d a recu le signal %d\n",  fils_termine, WTERMSIG(wstatus)) ;
            nb_fils_termines++ ;
         }
         else if (WIFCONTINUED(wstatus)) {
            printf("\nMon fils de pid %d a ete relance \n",  fils_termine) ;
         }
         else if (WIFSTOPPED(wstatus)) {
            printf("\nMon fils de pid %d a ete suspendu \n", fils_termine) ;
         }
         sleep(1) ;
      }
   }

}

/* Programme principal : Un pere qui cree 3 fils */

int main()
{
   int fils, pid ;
   int duree_sommeil = 2 ;

   signal(SIGCHLD, handler_chld) ;

   printf("Je suis le processus principal %d, de pere %d\n", getpid(), getppid()) ;
   nb_fils_termines = 0 ;

   for (fils=1 ; fils<=NB_FILS ; fils++) {
      pid = fork() ;
      if (pid<0) { // erreur fork
            printf("Erreur fork\n") ;
            exit(1) ;
```

```
58       }
59       else if (pid==0) { //fils
60             printf("\n     Processus fils num %d, de pid %d, de pere %d.\n", fils, getpid(), getppid())
61             if (fils == 4) {
62                duree_sommeil = 300 ;
63             }
64             sleep(duree_sommeil) ;
65             printf("\n     Processus fils num %d termine\n", fils) ;
66             exit(fils) ;  /* normalement 0, mais on veut illustrer WEXITSTATUS */
67       }
68       else {//pere
69             printf("\nProcessus pere de pid %d a cree un fils numero %d, de pid %d \n", getpid(), fils,
70       }
71    }
72
73    /* faire ce qu'on jusqu'à la terminaison de tous les fils */
74     do {
75         /* faire ce qu'on veut */
76
77     } while (nb_fils_termines < NB_FILS) ;
78    printf("\nProcessus Principal termine\n") ;
79    return EXIT_SUCCESS ;
80 }
```

On peut tester le fonctionnement de ce programme dans l'outil "apisys", ou en local selon l'enchaînement suivant :

1. lancer son exécution dans un terminal T1, et noter la prise en compte de l'arrêt rapide des 3 premiers fils : arrêt normal avec exit
2. dans un second terminal T2, taper la commande kill -TSTP pid_fils_4, et on verra s'afficher dans le terminal T1 un message indiquant que le père a pris connaissance de la suspension de son fils par le signal TSTP
3. dans le second terminal T2, taper la commande kill -CONT pid_fils_4, et on verra s'afficher dans le terminal T1 un message indiquant que le père a pris connaissance de la relance de son fils par le signal CONT

2.7 SIGALRM et alarm

La primitive `unsigned int alarm(unsigned int nb_secondes)`

entraîne l'envoi du signal SIGALRM au processus appelant après un laps de temps de nb_secondes. La valeur nb_secondes est copiée dans une horloge interne pour être décrémentée toutes les secondes jusqu'à atteindre 0 et provoquer l'envoi du signal SIGALRM.

- Un second appel à la primitive provoque la réinitialisation de l'horloge avec la dernière valeur spécifiée
- La valeur de retour contient le temps restant dans l'horloge

La primitive sleep vue précédemment, utilise elle même SIGALRM. Donc l'utilisation des deux prmitives dans le même code peut conduire à des perturbations. Par exemple, un appel à sleep après avoir lancé alarm écrase la valeur d'horloge de alarm.

La primitive alarm peut donc nous permettre de générer un événement toutes les N secondes, afin d'exécuter un traitement répétitif.

On peut illustrer cela en reprenant le code précédent, et en y apportant les modifications suivantes :

- on utilise le signal SIGALRM et la primitive alarm pour aller scruter régulièrement (toutes les 2 secondes) les changements d'état des processus fils
- pour que alarm soit réarmé autant de fois que nécessaire, il faut appeler de nouveau la primitive alarm à chaque fois que l'on entre dans le handler.

Fonctionnelement, cette solution présente une différence par rapport à la précédente : elle peut prendre en compte le changement d'état d'un fils avec un retard pouvant atteindre la valeur passée en paramètre à alarm.

Cela donne le code suivant :

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* utilisation de SIGALRM pour prendre connaissance de la fin des fils */

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */
#include <sys/wait.h> /* wait */
#include <signal.h>   /* opérations sur les signaux */

#define NB_FILS 3     /* nombre de fils */
#define D_ALARM 10     /* durée pour alarm */

int nb_fils_termines ; /* variable globale car modifiée par le traitant */

/* Traitant du signal SIGCHLD */
void handler_sigalrm(int signal_num) {
    int fils_termine, wstatus ;

    if (signal_num == SIGALRM) {
        while ((fils_termine = (int) waitpid(-1, &wstatus, WNOHANG | WUNTRACED | WCONTINUED)) > 0) {
            if WIFEXITED(wstatus) {
                printf("\nMon fils de pid %d s'est arrete avec exit %d\n",  fils_termine, WEXITSTATUS(ws
                nb_fils_termines++ ;
            }
            else if WIFSIGNALED(wstatus) {
                printf("\nMon fils de pid %d a recu le signal %d\n",  fils_termine, WTERMSIG(wstatus)) ;
                nb_fils_termines++ ;
            }
            else if (WIFCONTINUED(wstatus)) {
                printf("\nMon fils de pid %d a ete relance \n",  fils_termine) ;
            }
            else if (WIFSTOPPED(wstatus)) {
                printf("\nMon fils de pid %d a ete suspendu \n", fils_termine) ;
            }
        }
    }
    /* relancer alarm car cela n'est pas fait automatiquement */
    alarm(D_ALARM) ;
    return ;
}

int main()
{
    int fils, retour ;
    int duree_sommeil = 3 ;

    nb_fils_termines = 0 ;
    /* associer traitant sigchld à SIGCHLD */
    signal(SIGALRM, handler_sigalrm) ;

    printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
    /* Vidange du tampon de sortie pour que le fils le récupère vide        */
    fflush(stdout) ;

    for (fils = 1 ; fils <= NB_FILS ; fils++) {
        retour = fork() ;

        /* Bonne pratique : tester systématiquement le retour des appels système */
        if (retour < 0) {   /* échec du fork */
            printf("Erreur fork\n") ;
            /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
            exit(1) ;
        }

        /* fils */
        if (retour == 0) {
```

```
67              printf("\n     Processus fils numero %d, de pid %d, de pere %d.\n",
68                      fils, getpid(), getppid()) ;
69              sleep(duree_sommeil * fils) ;
70              exit(fils) ;  /* normalement exit(0), mais on veut illustrer WEXITSTATUS */
71          }
72
73          /* pere */
74          else {
75              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
76                      getpid(), fils, retour) ;
77          }
78      }
79
82      alarm(D_ALARM) ;
83      /* faire ce qu'on veut jusqu'à la terminaison de tous les fils */
84      do {
85          /* faire ce qu'on veut */
86
87      } while (nb_fils_termines < NB_FILS) ;
88      printf("\nProcessus Principal termine\n") ;
89      return EXIT_SUCCESS ;
90  }
```

On peut tester le fonctionnement de ce programme dans l'outil "apisys", ou en local selon l'enchaînement suivant : lancer son exécution dans un terminal, et noter que :

- les 3 fils s'arrêtent respectivent au bout de 3, 6, et 9 secondes
- la prise en compte de ces arrêts par le père ne se fait qu'après 10 secondes

En réalité, il existe trois horloges différentes, associées aux signaux SIGALRM, SIGVTALRM et SIGPROF, et que l'on peut programmer de façon plus fine en secondes et en microsecondes. Elles seront étudiées en TP.

2.8 Masquage des signaux

Dans certaines parties critiques de code, on doit se protéger pour ne pas être interrompu par des siganux usuels. Cette protection peut être réalisée en masquant les signaux par le recours à la primitive `sigprocmask`

`int sigprocmask (int mode, const sigset_t *ens, sigset_t *anciens)`

qui a le fonctionnement suivant :

- retour = 0 si tout s'est bien passé, -1 si erreur
- paramètre mode : éfinit le type de mise en place du masque défini dans ens
  - mode = SIG_SETMASK : remplace le masque existant par ens (ensemble des signaux à masquer)
  - mode = SIG_BLOCK : ajoute au masque existant les signaux appartenant à ens
  - mode = SIG_UNBLOCK : retire du masque existant les signaux appartenant à ens
- anciens : contient, au retour, la liste des signaux bloqués avant l'appel à sigprocmask. Mis à NULL si on ne veut pas récupérer cette information.

Pour masquer et démasquer des signaux, on doit donc :

- utiliser un ensemble de signaux de type sigset_t
- l'initialiser à vide avec la pritive sigemptyset : `int sigemptyset(sigset_t *ens)`
- lui ajouter un signal avec la pritive sigaddset : `int sigaddset(sigset_t *ens, int sig)`

Dautres primitives permettent de manipuler les ensembles de signaux :

- `int sigismember(sigset_t *ens, int sig)`
- `int sigfillset(sigset_t *ens)` : remplit un ensemble avec tous les signaux

- int sigdelset(sigset_t *ens, int sig)

On reprend l'avant dernier programme (traitement des événements des fils dans handler_sigchld), et on le modifie de la façon suivante :

- on divise la boucle en fin du programme principal en deux parties :
  - une première partie de 10 secondes durant laquelle on se protège du signal SIGCHLD
  - une seconde de 2 secondes durant laquelle on autorise le traitement du signal SIGCHLD

Ce qui donne le code suivant :

```c
/* Exemple d'illustration des primitives Unix : Un père et ses fils */
/* masquage et démasquage de SIGCHLD  */

#include <stdio.h>    /* entrées sorties */
#include <unistd.h>   /* pimitives de base : fork, ...*/
#include <stdlib.h>   /* exit */
#include <sys/wait.h> /* wait */
#include <signal.h>   /* opérations sur les signaux */

#define NB_FILS 3     /* nombre de fils */

int nb_fils_termines ; /* variable globale car modifiée par le traitant */

/* Traitant du signal SIGCHLD */
void handler_sigchld(int signal_num) {
    int fils_termine, wstatus ;

    if (signal_num == SIGCHLD) {
        while ((fils_termine = (int) waitpid(-1, &wstatus, WNOHANG | WUNTRACED | WCONTINUED)) > 0) {
            if WIFEXITED(wstatus) {
                printf("\nMon fils de pid %d s'est arrete avec exit %d\n",  fils_termine, WEXITSTATUS(ws
                nb_fils_termines++ ;
            }
            else if WIFSIGNALED(wstatus) {
                printf("\nMon fils de pid %d a recu le signal %d\n",  fils_termine, WTERMSIG(wstatus)) ;
                nb_fils_termines++ ;
            }
            else if (WIFCONTINUED(wstatus)) {
                printf("\nMon fils de pid %d a ete relance \n",  fils_termine) ;
            }
            else if (WIFSTOPPED(wstatus)) {
                printf("\nMon fils de pid %d a ete suspendu \n", fils_termine) ;
            }
        }
    }
    return ;
}

int main()
{
    int fils, retour ;
    int duree_sommeil = 2 ;

    sigset_t ens_signaux ;

    sigemptyset(&ens_signaux) ;

    /* ajouter SIGCHLD à ens_signaux */
    sigaddset(&ens_signaux, SIGCHLD) ;

    nb_fils_termines = 0 ;
    /* associer traitant sigchld à SIGCHLD */
    signal(SIGCHLD, handler_sigchld) ;

    printf("\nJe suis le processus principal de pid %d\n", getpid()) ;
    /* Vidange du tampon de sortie pour que le fils le récupère vide        */
    fflush(stdout) ;

    for (fils = 1 ; fils <= NB_FILS ; fils++) {
```

```
60          retour = fork() ;
61
62          /* Bonne pratique : tester systématiquement le retour des appels système */
63          if (retour < 0) {   /* échec du fork */
64              printf("Erreur fork\n") ;
65              /* Convention : s'arrêter avec une valeur > 0 en cas d'erreur */
66              exit(1) ;
67          }
68
69          /* fils */
70          if (retour == 0) {
71              printf("\n     Processus fils numero %d, de pid %d, de pere %d.\n",
72                      fils, getpid(), getppid()) ;
73              sleep(duree_sommeil * fils) ;
74              exit(fils) ;  /* normalement exit(0), mais on veut illustrer WEXITSTATUS */
75          }
76
77          /* pere */
78          else {
79              printf("\nProcessus de pid %d a cree un fils numero %d, de pid %d \n",
80                      getpid(), fils, retour) ;
81          }
82      }
83
86      /* faire ce qu'on veut jusqu'à la terminaison de tous les fils */
87      do {
88          /* période durant laquelle on ne veut pas être embêté par SIGCHLD */
89          printf("\nProcessus de pid %d : Je masque SIGCHLD\n", getpid()) ;
90          /* masquer les signaux définis dans ens_signaux : SIGCHLD */
91          sigprocmask(SIG_BLOCK, &ens_signaux, NULL) ;
92          sleep(10) ;
93
94          /* période durant laquelle on peut traiter le signal SIGCHLD */
95          printf("\nProcessus de pid %d : Je démasque SIGCHLD\n", getpid()) ;
96          /* démasquer les signaux définis dans ens_signaux : SIGCHLD */
97          sigprocmask(SIG_UNBLOCK, &ens_signaux, NULL) ;
98          sleep(2) ;
99
100     } while (nb_fils_termines < NB_FILS) ;
101     printf("\nProcessus Principal termine\n") ;
102     return EXIT_SUCCESS ;
103 }
```

Selon le code dessus :

- le fils 1 doit se terminer environ 2 secondes après lelancement du programme
- le fils 2 doit se terminer environ 4 secondes après lelancement du programme
- le fils 2 doit se terminer environ 6 secondes après lelancement du programme

En exécutant ce programme, on peut observer que :

- le prcessus père est protégé du signal SIGCHLD durant une période de 10 secondes environ
- il ne prend connaissance de la fin de ses fils qu'après démasquage du signal SIGCHLD