centralelille

Département
de
Mathématiques et Informatique

Système et réseau : module n° 2
Séminaire n° 3
Signaux

Samir El Khattabi *et* Christian Vercauter

Version éditée le 24 mars 2020

Année 2019-2020

Table des matières

Introduction

Ce document est un recueil d'exercices complété de rappels de cours, qui a été réalisé à partir de polycopiés de cours et d'exercices créés par Étienne Craye, Samir El Khattabi et Christian Vercauter pour plusieurs enseignements effectués à EC-Lille et à IG2I Lens.

Il concerne le concept de signal comme élément de communication et de de synchronisation entre processus dans un système d'exploitation multitâche.

Les informations présentées ici, sont conformes aux standards `POSIX` de l'IEEE établis à partir de la fin des années 1980, complétés et périodiquement révisés depuis ; la dernière révision date de 2017 (IEEE Std 1003.1™-2017)

Les exemples pris et les exercices proposés ont été testés sur un système `Linux` installé sur un ordinateur d'architecture `x86-64` bits.

Ils font appel à :

- des compétences SHELL acquises dans le module 1 de cet électif ;
- des connaissances de base sur la programmation en langage C ;
- des compétences sur les outils de développement associés.

Le compilateur `Gcc`

L'éditeur de textes `Emacs`

L'éditeur de textes `vim`

Dans la version avec réponses aux questions, de ce document, les solutions présentées ont été obtenues à partir de plusieurs ordinateurs dans différentes configurations :

`Ubuntu 18.04 LTS` en mode virtuel avec `VirtualBox`

`Ubuntu 18.04 LTS` en mode natif

`Debian 8.10` en mode natif

Chapitre I

Présentation générale

Les signaux constituent un des mécanismes de communication entre processus.

Le système LINUX prend en charge à la fois :

- les signaux POSIX classiques spécifiés dans le standard `IEEE Std 1003.1-1988`
- les signaux POSIX temps-réel spécifiés dans le standard `IEEE Std 1003.1b-1993`

La suite de ce document concerne essentiellement l'étude et l'utilisation des signaux POSIX classiques.

La commande `man 7 signal` dresse un panorama général des concepts et des usages des signaux sous LINUX. Quelques-uns de ces éléments sont présentés ci-dessous.

1. Définition

1.1 Notion d'interruption

La notion de signal est assez proche de la notion d'interruption abordée dans l'étude du fonctionnement d'un microprocesseur ou d'un microcontrôleur.

Dans ce contexte, une interruption est un événement interne ou externe au processeur qui peut provoquer l'interruption du programme en cours afin qu'un traitement spécifique à l'événement soit effectué. Lorsque ce traitement se termine, le programme interrompu, reprend son exécution à l'endroit précis et dans l'état précis dans lequel il se trouvait au moment de l'interruption.

Un événement interne peut par exemple être l'exécution d'une instruction illégale, la détection d'une division par zéro, un débordement de pile …

Une interruption externe correspond à la détection d'un changement d'état d'une broche du processeur, provoqué par un contrôleur de périphériques, un contrôleur programmable d'interruptions `PIC Intel 8259`, sur les premiers PC par exemple.

Sur un poste de travail de type PC, les demandes d'interruption externes ont de multiples sources :

- le circuit d'horloge, qui détermine la période de base utilisée par le noyau, notamment l'ordonnanceur afin de définir le quantum de temps attribué aux processus ;
- des périphériques comme le clavier, la souris ;
- des ports de communications ;
- des contrôleurs de disques, etc.

Pour un microcontrôleur, les sources d'interruption sont encore plus variées car directement liées aux équipements connectés : centrale inertielle, bus de communication industrielle, détecteur d'obstacles, capteurs de présence, de distance, de température, convertisseur analogique-numérique …

De façon générale :

a. le traitement d'une demande d'interruption (`IRQ` pour *Interrupt Request*) peut être autorisé ou inhibé ; dans ce dernier cas on dit que l'interruption est masquée ;

   Certaines interruptions sont non masquables (`NMI`, *Non Maskable Interrupt*).

b. les interruptions sont vectorisées ; à chaque source d'interruption correspond l'adresse du traitement réflexe qui lui est associée.

   Lorsqu'une demande d'interruption apparaît et si elle est validée, le traitement effectué par le processeur consiste à :

   1. sauvegarder certains registres du processeur dont le compteur ordinal [1], pour la sauvegarde de l'adresse de reprise, le mot d'état [2] souvent appelé `PSW` pour *Program Status Word*) ;
   2. l'effacement de la demande d'interruption signalant au processeur que celle-ci est prise en compte, c'-à-d en cours de traitement ;
   3. exécuter les instructions situées à partir de l'adresse associée à l'interruption, jusqu'à exécuter une instruction spécifique, comme *return from interrupt* [3] qui signale au processeur que le traitement est terminé, restaure les registres sauvegardés permettant ainsi la reprise du programme interrompu.

c. Une interruption a généralement un niveau de priorité de façon à traiter correctement les problèmes suivants :

   – Quand plusieurs demandes d'interruption sont présentes à un instant donné, laquelle doit être traitée immédiatement ?
   – Peut-on systématiquement interrompre, le traitement d'un interruption ?

   Pour la seconde question, la règle qui s'applique est la suivante : il n'y a interruption du traitement en cours que si la nouvelle demande d'interruption est strictement plus prioritaire que celle en cours de traitement.

   Les niveaux de priorité sont souvent programmables.

1.2 Notion de signal

Un signal est une interruption logicielle qui correspond à :

- un événement interne déclenché par exemple par une erreur détectée par l'unité de calcul ou l'unité de gestion de la mémoire, lors de l'exécution d'un calcul de logarithme, de l'accès à un élément d'un tableau ou de l'accès à la variable désignée par un pointeur.

1. `rip` : registre pointeur d'instruction, pour les processeurs de la famille `x86-64`
2. `rflags` : registre des indicateurs, pour les processeurs de la famille `x86-64`
3. `iretq` pour les processeurs de la famille `x86-64`

- un événement externe déclenché par une commande, un programme ou par une action réalisée au clavier par l'utilisateur.

  Exemples :

  1. Un utilisateur exécute la commande permettant l'arrêt d'un processus donné ;
  2. L'utilisateur effectue un `Ctrl-c` ou un `Ctrl-z` ;
  3. un processus indique à un autre processus qu'il vient d'ajouter une nouvelle information dans un fichier dont il partage l'usage ;
  4. un processus a programmé l'exécution d'un traitement donné à un instant précis.

Comme pour les interruptions évoquées précédemment, un signal peut être masqué. S'il est autorisé — certains d'entre-eux le sont toujours — il provoque l'interruption du programme en cours, afin que le traitement associé soit exécuté ; il y a un traitement par défaut pour chaque signal. Lorsque le traitement est terminé, le programme reprend son exécution normale.

Il n'y a pas de notion de priorité pour les signaux classiques : le traitement d'un signal peut lui même être interrompu par un autre signal.

Un signal envoyé à un processus bloqué ou à un processus dans l'état `Ready` est enregistré dans son `BCP`, et ne sera traité que lorsque celui-ci passera dans l'état `Run`.

1.3 États d'un signal

Un signal peut être dans l'un des états suivants :

***émis* :** ou *envoyé* mais pas encore pris en compte ;

***délivré* :** ou *pris en compte* pour indiquer que l'action associée est en cours de traitement ou terminée ;

***pendant* :** le signal émis n'a pas encore été pris en compte car le processus destinataire est dans l'état prêt ou dans l'état bloqué ;

Une seule occurrence d'un même type de signal peut être pendante ; cela signifie que, si un processus reçoit trois signaux de même type pendant qu'il est dans l'état *bloqué*, il n'effectuera qu'un seul traitement lorsqu'il passera dans l'état *Run*

***masqué* :** la prise en compte du signal est volontairement différée ; il ne sera traité que lorsque le processus destinataire l'autorisera.

Le noyau gère pour chaque processus une table des états de signaux, le masque des signaux, et une table des vecteurs de signaux (fonctions de traitement). Ces informations font partie du `BCP` d'un processus.

2. Liste des signaux

Sous UNIX et dérivés les signaux sont désignés par un numéro et par un nom symbolique préfixé par `SIG`. Ils sont au nombre de 64 et se répartissent en

- 32 signaux classiques numérotés de 1 à 32 (certains d'entre-eux sont obsolètes)
- au plus 32 signaux temps réels, utilisant les numéros qui suivent.

Question 1.

Quelle est la commande permettant d'afficher la liste des signaux pris en charge sur votre système UNIX ou dérivés.

Indication. Consulter l'aide en ligne de la commande `kill`

Il faut cependant noter que les numéros associés à certains signaux peuvent être différents selon les architectures de système UNIX, comme le signal `SIGSTOP` qui permet d'arrêter un processus et porte le numéro :

– 17 sur les architectures `Alpha` (*Digital Equipment Corporation*) et `Sparc` (*Sun microsystems*)

– 19 pour la majorité des architectures dont les `x86`, `x86-64` et `ARM` ;

– 23 pour les architectures `Mips`.

Il est par conséquent conseillé d'utiliser les noms symboliques des signaux plutôt que leur numéro.

Question 2.

1. Quelle est la commande permettant d'afficher **uniquement** le numéro du signal `SIGTSTP` ?
2. Quelle est la commande permettant d'afficher **uniquement** le nom symbolique du signal n° 10 ?

Parmi tous les signaux classiques ($n^o \in [1, 32]$), seuls les signaux `SIGUSR1` et `SIGUSR2` peuvent être utilisés pour des besoins spécifiques du programmeur ; les autres ayant une fonction prédéfinie.

Question 3.

1. Que représente le signal `SIGINT` ? Citer un moyen simple de produire ce signal.
2. Que représente le signal `SIGCHLD` et dans quelle circonstance, est-il généré ?
3. Que représente le signal `SIGTSTP` et dans quelle circonstance, est-il généré ?

Chapitre II

Traitement des signaux

1. Commande d'envoi de signal

Le rôle principal de la commande `kill` est d'envoyer un signal à un ou plusieurs processus.

Le *choix contestable du nom* de cette commande provient de l'effet réalisé par défaut, par la réception de la plupart des signaux sur un processus : celui-ci se termine !

Lorsqu'on utilise cette commande on désigne le signal envoyé par son numéro ou par le nom symbolique associé sans le préfixe `SIG` puis on énumère les processus destinataires

Exemple. `kill -INT 5437 5441 5448` qui équivaut à `kill -2 5437 5441 5448` sur un système LINUX pour une architecture `x86-64`.

2. Traitement par défaut d'un signal

La délivrance d'un signal entraîne l'exécution d'une fonction particulière appelée gestionnaire de signal(*signal handler*)

Selon le signal délivré, le gestionnaire par défaut effectue l'une des actions suivantes :

- Le processus se termine.

  C'est l'action par défaut, définie par exemples pour les signaux `SIGTERM`, `SIGUSR1`, `SIGALRM` … ;
- une image mémoire est générée (*core dumped*) permettant une analyse *post-mortem* du processus puis celui-ci se termine.

  C'est le traitement par défaut des signaux `SIGILL`, `SIGSEGV`, `SIGXCPU` …
- rien, le signal reçu est simplement ignoré ;

  C'est le cas par exemples, des signaux `SIGCHLD`, `SIGURG`
- le processus est arrêté (suspendu) ;

  C'est le traitement par défaut des signaux `SIGSTOP`, `SIGTSTP`
- le processus s'il était arrêté, peut reprendre son exécution, comme dans le cas du signal `SIGCONT`

Question 4.

Soit un programme présenté ci-dessous, dont la fonction `main()` est constituée de l'instruction

```
while(1) ;
```

Après avoir exécuté la commande `ulimit -c unlimited` qui autorise la création de fichier d'image de processus, exécuter ce programme en arrière-plan et noter le comportement obtenu, lorsque qu'il reçoit les signaux suivants :

- `SIGINT`,
- `SIGCONT`,
- `SIGUSR1`
- `SIGFPE`

Le code de ce programme `infini.c`, est celui-ci :

```c
int main(void)
{
  while (1);
  return 0;
}
```

3. Quelques appels système

3.1 L'appel système `kill()`

Le rôle de l'appel système `kill()`, à l'instar de la commande de même nom, n'est pas de *tuer* un processus mais de lui envoyer une signal, ce qui dans bien des cas, revient au même.

Question 5.

a) Donner la commande permettant d'obtenir de l'aide sur l'appel système `kill()`

b) Est-il conforme au standard `POSIX` ?

c) Permet-il d'envoyer un même signal à plusieurs processus ?

d) Quelle information sur le processus 1234 obtient-on en exécutant l'opération

```c
int resultat = kill(1234, 0);
```

3.2 Appel système `alarm()`

Question 6.

Quelle est la commande permettant d'obtenir de l'aide sur l'appel système `alarm()`

Quelle est sa fonction ?

3.3 Appel système `pause()`

Question 7.

Quelle est la commande permettant d'obtenir de l'aide sur l'appel système `pause()`

Quelle est sa fonction ?

3.4 Application nº 1

Question 8.

Créer une copie du programme `infini` que vous nommerez `infini_v2` dans laquelle l'instruction `while(1);` est remplacée par `alarm(10); pause();`

Compiler puis exécuter ce programme et expliquer le comportement de cette application.

---

4. Ignorer ou masquer un signal

4.1 Désigner les signaux dans un programme `C`

Nous avons vu précédemment qu'il était utile et parfois nécessaire d'inclure le fichier `signal.h` en tête de code source pour disposer des définitions des noms de signaux, des types relatifs et des prototypes de fonctions, relatifs aux signaux.

Question 9.

a) Utiliser la commande `locate` pour découvrir toutes les occurrences du fichier `signal.h`

b) Parmi les résultats obtenus, quel est le fichier utilisé par le compilateur `gcc` pour la directive `#include <signal.h>`. Justifier votre réponse.

c) Ce fichier contient-il les définitions des constantes symboliques désignant les signaux ?

d) Quel est le fichier contenant réellement ces définitions ?

---

5. Installer un gestionnaire de signal

5.1 L'appel système `sigaction()`

L'installation d'un nouveau gestionnaire de signal est réalisée par l'appel système `sigaction()`

La commande `man 2 sigaction` permet d'obtenir des informations complémentaires sur cet appel système.

```
SIGACTION(2)            Manuel du programmeur Linux              SIGACTION(2)

NOM
       sigaction - Examiner et modifier l'action associée à un signal

SYNOPSIS
       #include <signal.h>
```

```
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

Le paramètre `signum` désigne le signal pour lequel on souhaite installer un autre gestionnaire. Il n'est pas permis d'installer un gestionnaire pour les signaux `SIGKILL` et `SIGSTOP` ; il n'est pas, non plus, permis de les masquer.

Les deuxième et troisième paramètres de l'appel système `sigaction()` sont des pointeurs de structure de type `struct sigaction`

Exemple d'utilisation :

```
struct sigaction newAction, oldAction;
// ... ici on renseigne les composantes de la structure newAction

CHECK(sigaction(SIGUSR2, &newAction, &oldAction), "sigaction()");
```

On récupère dans la structure `oldAction` les paramètres de l'ancienne action associée au signal `SIGUSR2` S'il n'est pas utile de récupérer les paramètres de l'ancienne action alors on peut remplacer l'instruction de la ligne 4 ci-dessus, par :

```
CHECK(sigaction(SIGUSR2, &newAction, NULL), "sigaction()");
```

a) La structure `sigaction` : 1ère version

La structure[1] `sigaction` est définie comme suit :

```
struct sigaction {
    void     (*sa_handler)(int);
    sigset_t   sa_mask;
    int        sa_flags;
};
```

La première composante nommée `sa_handler` est un pointeur sur une fonction qui ne retourne rien et qui a un paramètre de type entier : en pratique on range dans ce paramètre l'adresse de la fonction chargée de traiter un ou plusieurs signaux ; cette fonction lorsqu'elle est appelée, reçoit l'identité du signal à traiter.

Le second paramètre, nommé `sa_mask` définit les signaux supplémentaires à bloquer, pendant l'exécution du nouveau gestionnaire. Le signal ayant activé le gestionnaire, est bloqué pendant son exécution.

Le dernier paramètre, `sa_flags` est formé d'une composition d'attributs reliés par des `OU binaires` (|). Parmi ces attributs, il y a l'attribut `SA_RESTART` qui permet de redémarrer automatiquement certains appels système interrompus par l'arrivée du signal.

b) Gestionnaires prédéfinis de signaux

Il existe deux gestionnaires de signaux prédéfinis :

1. Une structure en `C` est comparable à une classe Java sans aucune méthode et dont les composantes sont toutes publiques. On crée une variable structure comme dans cet exemple : `struct sigaction action;`
On accède à ses composantes en utilisant la notation pointée : `action.sa_flags = 0;`

- `SIG_DFL` qui permet de rétablir le traitement par défaut d'un signal ;
- `SIG_IGN` qui permet d'*ignorer le signal*, c'est-à-dire traiter le signal délivré en ne faisant rien.

Bloquer un signal et ignorer un signal ne sont pas tout à fait équivalents :

– dans le premier cas, le signal bloqué reste dans la liste les signaux à traiter jusqu'à ce qu'il soit débloqué ;

– dans le second cas, il est extrait de cette liste, car il est considéré comme traité, même si le traitement se réduit à ne rien faire.

Question 10.

– Créer une copie du programme `signal1.c`, nommée `signal2.c`

– Supprimer ou mettre en commentaires, les opérations qui permettent de masquer le signal `SIGINT`

– Mettre à la place des opérations précédentes, les opérations qui définissent le traitement `SIG_IGN` du signal `SIGINT`

c) Gestionnaire de signaux

C'est une fonction qui ne retourne pas de résultat et qui a un paramètre servant à identifier le signal à traiter.

Il est fréquent d'utiliser le même gestionnaire pour différents signaux mais on peut aussi définir un gestionnaire distinct pour chaque signal à traiter.

En langage `C`, le nom d'une fonction, lorsqu'il n'est pas suivi d'une paire de parenthèses, désigne son adresse de début.

Exemple de structure d'un gestionnaire des signaux `SIGCHLD`, `SIGUSR1`, `SIGUSR2` et `SIGALRM` :

```
static void signalHandler(int numSig)
{
    switch (numSig)
    {
    case SIGCHLD:     /*  traitement de SIGCHLD  */
        break;

    case SIGUSR1:     /*  traitement de SIGUSR1  */
        break;

    case SIGUSR2:     /*  traitement de SIGUSR2  */
        break;

    case  SIGALRM:    /*  traitement de SIGALRM  */
        break;

    default:
        printf("Signal %d non traité\n", numSig);
        break;
    }
}
```

5.2 Application n° 3

Question 11.

Créer une copie du programme `signal2.c`, nommée `signal3.c` puis apporter les modifications permettant d'afficher le message *Le contrôle-C est désactivé*, chaque fois que l'utilisateur tape `Ctrl-c` au clavier.

Question 12.

a) Que constate-t-on lors de l'exécution de ce programme ?

b) Quelle en est la cause ?

c) Apporter les modifications permettant de corriger ce problème.

Question 13.

Créer une copie du programme précédent, nommée `signal4.c` puis lui apporter les modifications suivantes :

- le programme commence par afficher le message suivant :

  Le Ctrl-c est désactivé pendant `n` *s.* où n est la durée transmise comme argument sur la ligne de commande (10 s par défaut)

- Pendant la période de *désactivation du Ctrl-c* chaque `Ctrl-c` tapé par l'utilisateur provoque l'affichage d'un message se présentant comme suit :

  Le Ctrl-c est désactivé pendant `m` *s.* où m représente la durée restante de désactivation

- À l'issue de la période de désactivation, le message ci-dessous est affiché :

  Le traitement normal du Ctr-l est réactivé et une action sur `Ctrl-c` permet désormais d'arrêter le programme.

Exemple de résultat à obtenir :

```
$ ./signal4 8
Le Ctrl-c est désactivé pendant 8 s.

^C     Le Ctrl-c est désactivé pendant 7 s.
^C    Le Ctrl-c est désactivé pendant 4 s.
^C    Le Ctrl-c est désactivé pendant 1 s.
Le traitement normal du Ctr-c est réactivé
^C
$
```

Indication.

- le gestionnaire de signaux doit également traiter le signal `SIGALRM` ;
- définir la variable `timeout` comme étant globale, de façon à pouvoir y accéder dans le gestionnaire de signaux ;
- utiliser `SIG_DFL` afin de rétablir le traitement initial du signal `SIGINT`

6. Signaux et processus

6.1 Cas du `fork()`

Un processus fils créé par l'exécution de l'appel système `fork()` hérite de la gestion des signaux préalablement mis en place par son père. Le processus fils hérite :

- d'une copie du masque des signaux ; les signaux masqués par le père le sont aussi pour son fils ;
- d'une copie de la liste des signaux pendants ;les signaux bloqués par le père le sont aussi pour le fils ;
- de la la mise en place de gestionnaires ; les gestionnaires mis en place par le père, sont aussi les gestionnaires des signaux du fils ;

a) Préparation de l'exemple de vérification

Afin de vérifier le comportement décrit ci-dessus, procéder comme suit :

a. Créer une copie du programme précédent et la nommer `signal5.c` ;

b. Insérer les opérations permettant de créer un processus fils, juste après avoir installé le gestionnaire des signaux `SIGINT` et `SIGALRM` ;

c. Ajouter dans le corps du processus fils, les opérations suivantes :

```
    case 0 :
        /*  On met ici le code spécifique du processus fils */
        CHECK(alarm(1), "alarm()");
        while (pause() == -1 && errno == EINTR);
        exit (0); /* Fin normale du fils  */
```

d. Faites en sorte que le processus père affiche le `PID` du fils qu'il a créé puis se termine.

Le code complet de cette nouvelle version, est le suivant :

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

#define TIMEOUT 10

#define CHECK(sts, msg)             \
    if (-1 == (sts)) {              \
        perror(msg);                \
        exit(EXIT_FAILURE);         \
    }

static void  signalHandler (int);

int timeout = 0;

int main (int argc, char *argv[])
{
    struct sigaction newAction;
```

```
    pid_t   pidFils;

    if (argc > 1) {
        timeout = atoi(argv[1]);
    }
    if (timeout <= 0)
        timeout = TIMEOUT;

    printf("Le Ctrl-c est désactivé pendant %d s.\n\n", timeout);

    /* Initialisation de la structure sigaction                    */
    newAction.sa_handler = signalHandler;
    CHECK(sigemptyset(&newAction.sa_mask), "sigemptyset()");
    newAction.sa_flags = 0;
    /* Installation du gestionnaire du signal SIGINT               */
    CHECK(sigaction(SIGINT, &newAction, NULL), "sigaction()");
    CHECK(sigaction(SIGALRM, &newAction, NULL), "sigaction()");

    pidFils = fork();
    switch (pidFils) {
    case -1 :
        perror("Échec de la création d'un processus fils");
        exit (pidFils);

    case 0 :
        /*  On met ici le code spécifique du processus fils */
        CHECK(alarm(1), "alarm()");
        while (pause() == -1 && errno == EINTR);
        exit (0); /* Fin normale du fils  */

    default :
        /* On met ici la suite du code du processus père */
        printf("Je suis le père du processus ..... n°%d\n", pidFils);
        break;
    }
    exit(EXIT_SUCCESS);
}

static void signalHandler(int numSig)
{
    switch (numSig) {
    case SIGINT:    /*  traitement de SIGINT   */
        printf("\tLe Ctrl-c est désactivé pendant %d s.\n", timeout);
        break;

    case SIGALRM:    /*  traitement de SIGALRM   */
        timeout--;
        if (timeout <= 0) {
            struct sigaction newAction;
            /* Initialisation de la structure sigaction             */
            newAction.sa_handler = SIG_DFL; /* gestionnaire par défaut*/
            CHECK(sigemptyset(&newAction.sa_mask), "sigemptyset()");
            newAction.sa_flags = 0;
            /* Installation du gestionnaire par défaut              */
            CHECK(sigaction(SIGINT, &newAction, NULL), "sigaction()");
            printf("Le traitement normal du Ctr-c est réactivé\n");
       }
        else
            CHECK(alarm(1), "alarm()");
        break;

    default:
        printf("Signal %d non traité\n", numSig);
        break;
    }
}
```

b) Exemple de tests

Après avoir généré le programme exécutable `signal5`, effectuer les opérations suivantes :

- lancer l'exécution du programme pour une durée de l'ordre de 20 s ;
- dès que le processus père se termine, dans la fenêtre *terminal* effectuer une série de commandes `kill -INT <pid du processus fils>`

c) Exemple de résultat

```
./signal5 20
Le Ctrl-c est désactivé pendant 20 s.

Je suis le père du processus ..... n°2507
$ kill -INT 2507
        Le Ctrl-c est désactivé pendant 13 s.
$ kill -INT 2507
        Le Ctrl-c est désactivé pendant 11 s.
$ kill -INT 2507
        Le Ctrl-c est désactivé pendant 7 s.
$ kill -INT 2507
        Le Ctrl-c est désactivé pendant 3 s.
Le traitement normal du Ctr-l est réactivé
$
```

Cet exemple montre bien que le processus fils a hérité de la gestion des signaux mis en place par son père, avant sa naissance.

Naturellement les `Ctrl-c` effectués au clavier par l'utilisateur après la fin de processus père, ne sont pas envoyés au processus fils mais envoyés à l'interpréteur de commande, père de `signal5`

Il est donc nécessaire d'envoyer le signal `SIGINT` au processus fils, à l'aide de la commande `kill`.

Question 14.

Ajouter avant le `return(EXIT_SUCCESS);` présent dans la fonction `main()` du code du programme précédent, l'instruction `CHECK(wait(&status), "wait()");` Cette opération demande que le processus père attende la terminaison de son fils, avant de s'arrêter lui-même.

Compléter également les opérations d'affichage de façon à afficher le `pid` du processus qui les exécute.

Comment expliquer les résultats obtenus par l'exécution de cette nouvelle version du programme nommée `signal5b`, présentés ci-dessous :

```
$ ./signal5b 30
Le Ctrl-c est désactivé pendant 30 s.

Je suis le père du processus ..... n°1563
^C      [1563] --> Le Ctrl-c est désactivé pendant 28 s.
        [1562] --> Le Ctrl-c est désactivé pendant 30 s.
wait(): Interrupted system call
$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  1210  1205  0  80   0 -  5934 -      pts/1    00:00:00 bash
1 S  1000  1563     1  0  80   0 -  1021 -      pts/1    00:00:00 signal5b
0 R  1000  1564  1210  0  80   0 -  2674 -      pts/1    00:00:00 ps
```

```
$ kill -INT 1563
     [1563] --> Le Ctrl-c est désactivé pendant 12 s.
$ kill -INT 1563
     [1563] --> Le Ctrl-c est désactivé pendant 8 s.
$ kill -INT 1563
     [1563] --> Le Ctrl-c est désactivé pendant 2 s.
[1563] --> Le traitement normal du Ctr-c est réactivé
$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  1210  1205  0  80   0 -  5934 -      pts/1    00:00:00 bash
1 S  1000  1563     1  0  80   0 -  1021 -      pts/1    00:00:00 signal5b
0 R  1000  1568  1210  0  80   0 -  2674 -      pts/1    00:00:00 ps
$ kill -INT 1563
$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  1210  1205  0  80   0 -  5934 -      pts/1    00:00:00 bash
0 R  1000  1569  1210  0  80   0 -  2674 -      pts/1    00:00:00 ps
$
```

Question 15.

Comment peut-on remédier au problème observé lors du test précédent ?

6.2 Cas de `execve()`

Lorsqu'un processus fils qui a hérité de la gestion des signaux de son père, exécute un appel système `execve()`, tous ses attributs sont préservés (`pid`, `ppid` ...) à l'exception des gestionnaires de signaux qui sont tous réinitialisés à `SIG_DFL`

Les masques de signaux préalablement définis, restent toujours actifs ; l'ensemble de signaux en attente (reçu mais non traité car masqué) est conservé.

Pour vérifier ce comportement, il suffit d'adapter le code du programme précédent, de la façon suivante :

1. le traitement du processus fils (cas où l'appel système `fork()` retourne 0), comme suit :

```
    case 0 :
        processusFils(timeout);
        exit (0); /* exit() de précaution -> Fin normale du fils  */
```

2. le processus fils reçoit la valeur du *time out* et exécute le programme `signal6_Fils`

```
#define PROCESSFILS_NAME    "signal6_Fils"
/* ... */
void processusFils(int tim_out)
{
    char sTimeOut [11];
    char *args[] = {PROCESSFILS_NAME, sTimeOut, NULL};
    sprintf(sTimeOut, "%d", tim_out);
    CHECK(execve(PROCESSFILS_NAME, args, NULL), "execve()");
}
```

3. le programme `signal6_Fils` réalise les opérations qui étaient effectuées par le processus fils de la version précédente, à savoir :

```c
int main (int argc, char *argv[])
{
    int timeout = 0;
    if (argc > 1)
        timeout = atoi(argv[1]);
    if (timeout <= 0)
        timeout = TIMEOUT;

    /*  On met ici le code spécifique du processus fils */
    CHECK(alarm(timeout), "alarm()");
    while (pause() == -1 && errno == EINTR);
    exit (0); /* Fin normale du fils  */
}
```

On vérifie alors que le processus fils réagit de manière standard aux signaux `SIGINT` et `SIG_ALRM` : il est interrompu.

On peut aussi vérifier en ajoutant le masquage du signal `SIGINT` avant la création du processus fils, que ce signal est également masqué pour le programme `signal6_Fils` lancé par un `execve()`

```c
/* ... */
#ifdefined (_WITH_INTMASK)
    /* Initialisation du masque des signaux à ajouter                    */
    CHECK(sigemptyset(&newMask), "sigemptyset()");

    /* Ajout du signal SIGINT                                            */
    CHECK(sigaddset(&newMask, SIGINT), "sigaddset(SIGINT)");
    /* Ajout dans le masque et sauvegarde de l'ancien masque             */
    CHECK(sigprocmask(SIG_BLOCK, &newMask, &oldMask), "sigprocmask()");
#endif
    pidFils = fork();
/* ... */
```

Chapitre III

Exercice de synthèse

1. Cahier des charges

1.1 Objectif

Un processus qui ne peut être interrompu par un `contrôle-C`, crée un processus fils puis arme une alarme.

Le processus fils invite l'utilisateur à entrer un mot de passe dans un délai fixé et avec un nombre de tentatives limité. Il affiche également un message signalant que le contrôle-C est désactivé lorsque l'utilisateur tape contrôle-C.

Le processus fils réalise la lecture et le contrôle du mot de passe saisi par l'utilisateur et signale à son père la raison de sa terminaison :

- épuisement des tentatives ;
- saisie valide du mot de passe.

Le processus père affiche un message de dépassement de délai lorsque le temps imparti au processus fils expire ; il arrête alors le processus fils.

Dans le cas contraire, il signale que le nombre maximal d'essais a été atteint ou que le mot de passe saisi est correct.

1.2 Spécification détaillée

Le mot de passe est une chaîne d'au plus 8 caractères.

Le processus père … :

- … récupère sur la ligne de commande :
  1. la durée maximale accordée pour la saisie du mot de passe. Par défaut cette durée est égale à 30 s.
  2. le nombre maximal d'essais de saisie du mot de passe. Par défaut ce nombre est égal à 3.
- … transmet à son fils le nombre d'essais de saisie de mot de passe ;
- … met fin à l'exécution de son fils lorsque la durée maximale de saisie est atteinte.

Le processus fils se termine en fournissant comme code de terminaison :

- la valeur 0 pour signaler la validité du mot de passe saisi
- le nombre d'échecs subis, qui doit être égal au nombre de tentatives autorisées.

1.3 Exemples de comportement

Dans les exemples ci-dessous le mot de passe valide est *essai*

a) Avec paramètres, un Ctrl-c, un échec, un succès

```
$ ./signal7  15 4
Vous avez 15 secondes pour entrer votre mot de passe
Je suis le père du processus ..... n°4257
Vous avez 4 essais pour entrer votre mot de passe, et une durée limitée
Premier essai .... : ^C [4257] --> Le Ctrl-c est désactivé.
erttyyuu
2ème essai ....... : essai

Terminaison du processus 4257   Mot de passe valide : connexion acceptée
[4256] Fin du processus père
```

b) Sans paramètre, deux Ctrl-c et limite de durée atteinte

```
$ ./signal7
Vous avez 30 secondes pour entrer votre mot de passe
Je suis le père du processus ..... n°4292
Vous avez 3 essais pour entrer votre mot de passe, et une durée limitée
Premier essai .... : azerty
2ème essai ....... : sesame
Dernier essai .... : ^C [4292] --> Le Ctrl-c est désactivé.
^C      [4292] --> Le Ctrl-c est désactivé.

Terminaison du processus 4292   Délai expiré : connexion refusée
[4291] Fin du processus père
```

c) Sans paramètre et trois échecs

```
$ ./signal7
./signal7
Vous avez 30 secondes pour entrer votre mot de passe
Je suis le père du processus ..... n°4302
Vous avez 3 essais pour entrer votre mot de passe, et une durée limitée
Premier essai .... : azerty
2ème essai ....... : qsdfgh
Dernier essai .... : wxcvbn

Terminaison du processus 4302   Échec des 3 tentatives : connexion refusée
[4301] Fin du processus père
```

2. Indications

2.1 Signalisation père → fils

Le processus père demande la terminaison du processus fils en lui envoyant un signal (`SIGUSR2` par exemple)

2.2 Attente et analyse de terminaison du fils

Le père attend la terminaison de son fils et doit en déterminer la cause. Plusieurs possibilités existent :

a) le fils se termine parce qu'il a reçu le signal de son père

b) le fils se termine parce que le bon mot de passe a été donné

c) le fils se termine parce que toutes les tentatives ont échoué.

La distinction entre tous ces cas est réalisée à l'aide des macro-fonctions `WIFEXITED`, `WIFSIGNALED`, `WIFEXITSTATUS` et `WTERMSIG`.

Consulter le manuel de l'appel système `wait()` pour découvrir ce qu'elles font et comment les utiliser.

2.3 Saisie du mot de passe

Pour ne pas compliquer le traitement, le mot de passe saisi par l'utilisateur, apparaît en clair sur l'écran.

La lecture des `MAXLEN` premiers caractères tapés au clavier par l'utilisateur et leur rangement dans une chaîne de caractères peut être réalisée comme suit :

```
1 char mdp [MAXLEN + 1];  /* la variable recevant la chaîne saisie        */
2 char *p;
3 /* ... */
4 fgets(mdp, sizeof mdp, stdin);
5 if ( (p = strchr(mdp,'\n')) != NULL)
6    *p = '\0';
7 else
8     while (getchar() != '\n');
```

Dans l'exemple ci-dessus :

- la ligne 4, limite à `MAXLEN` le nombre de caractères rangés dans la variable `mdp`. La lecture est faite au clavier (`stdin`)
- la ligne 5, recherche la présence du caractère de fin de ligne (`'\n'`). S'il est présent alors il est remplacé par le caractère de fin de chaîne (`'\0'`) à la ligne 6, s'il est absent, c'est que l'utilisateur a tapé plus de `MAXLEN` caractères. Ces caractères sont simplement lus et ignorés à la ligne 8, jusqu'à ce qu'on atteigne la marque de fin de ligne : le buffer de clavier est maintenant vide.

2.4 Contrôle du mot de passe

Il suffit de comparer la chaîne de caractères saisie par l'utilisateur au mot de passe valide, codé en *dur* dans le code du programme, par souci de simplicité.

```
#define PASSWORD        "essai"
```

La comparaison entre chaînes de caractères est réalisée à l'aide de la fonction `strcmp()` qui retourne 0 lorsque les chaînes comparées sont égales, comme la méthode `compareTo()` de la classe `String` en `Java`.

Exemple :

```
    if (strcmp(mdp, PASSWORD) == 0)
          exit(EXIT_SUCCESS);
```

Il faut ajouter `#include <string.h>` en tête de programme.

2.5 Bilan des signaux utilisés

Signal	Père	Fils
SIGINT	masque mais installe le gestionnaire pour son fils	Démasqué, affichage d'un message
SIGALRM	installe le gestionnaire qui permet l'envoi de SIGUSR2	masque ce signal
SIGUSR2	signal envoyé à son fils	traitement par défaut : arrêt

Question 16.

Écrire le programme répondant au cahier des charges et aux spécifications présentées ci-dessus.

Bibliographie

[1] W. Richard Stevens, Stephen A. Rago *Advanced Programming in the UNIX Environment*, Addison-Wesley, Third Edition, 2013.

[2] Robert Love *Linux System Programming* O'Reilly, 2007.

[3] Christophe Blaess *Programmation système en C sous Linux* Eyrolles, 2e édition, 2005.

[4] Patrick Cegielski *Conception de systèmes d'exploitation : Le cas Linux* Eyrolles, 2e édition, 2004.

[5] Vincent Lozano *UNIX : Pour aller plus loin avec la ligne de commande* In Libro Veritas, Framabook, 2010 (Téléchargeable ici)

[6] Mendel Cooper, Linux Documentation Project *Advanced Bash-Scripting Guide*

- Version anglaise en ligne ;
- Traduction en français en ligne.

[7] Robert Mecklenburg *Managing Projects with GNU Make* O'Reilly, 3rd edition, 2004 (en ligne sur le projet Open Books)

[8] Richard M. Stallman and the GCC Developer Community *Using the GNU Compiler Collection* GNU Press

- Version anglaise à lire en ligne
- Version anglaise au format pdf