R3.05 - TP 2 - Signaux

Vous devez avoir compris l'exercice 8 du TP 1 sur **`argc`** et **`argv`**.

Si ce n'est pas le cas, terminez l'exercice 8 avant de commencer ce TP sur les signaux.

Introduction

Dans la vie quotidienne, un signal survient pour attirer l'attention sur un événement important (ou supposé l'être), peu importe ce qu'on est en train de faire : signal d'alarme incendie, signal du réveil matin, signal de la barrière automatique qu'un train va passer, signal du départ d'une course. On pourrait en trouver bien d'autres tant nous vivons dans un monde où les signaux sont omniprésents, ne serait-ce qu'avec nos smartphones qui génèrent des signaux en tout genre pour attirer notre attention.

Attirer l'attention est le but du signal. C'est ensuite à celui qui reçoit le signal de décider de la réaction appropriée : quitter la salle et aller au point de regroupement (signal incendie), se lever (réveil matin), arrêter son véhicule (signal SNCF), etc. Cependant, chacun est libre de faire la sourde oreille à la sonnerie du réveil ou d'ignorer le son de l'arrivée d'un tweet ou d'un post d'Instagram sur son smartphone (un cas très c'est rare, surtout chez les étudiant.e.s)

Généralités

Les processus sous Linux reçoivent aussi des signaux, pour les prévenir d'un événement qui peut les intéresser. Nous allons en voir certains.

A l'instar de l'être humain qui peut ignorer ces signaux, un processus peut aussi réagir comme il le souhaite sur la plupart des signaux qu'il reçoit.

Principe d'interruption d'exécution

Comme pour les signaux de notre vie quotidienne, un processus est interrompu à la réception d'un signal, peu importe ce qu'il était en train de faire. Il reprendra sa tâche quand il aura terminé de réagir au signal. On va voir dans quelles conditions et comment il procède.

Ce que le processus était en train de faire peut se résumer à 2 situations :

- Soit il était en cours d'exécution d'un bout de code à lui, comme par exemple un **`printf()`**, un calcul arithmétique, une boucle, etc. Quand il est en train d'exécuter du code qui lui appartient (qui fait partie de son code binaire, résultat de la compilation), on dit que le processus est dans le **User Space**.
- Soit il était en cours d'exécution d'un bout de code système, c'est-à-dire qu'il était dans l'attente du retour d'un appel système, un **`open()`**, un **`read()`**, un **`write()`**, un

`stat()` par exemple. Quand il est en train d'exécuter du code système, donc du code qui ne fait pas partie de son code binaire mais qui est fourni par le système (dans son noyau), on dit qu'il est dans le **Kernel Space**. Même si on appelle une fonction système dans notre programme, le code de cette fonction n'est pas intégré dans le binaire produit à la compilation, c'est uniquement l'appel qui est intégré, et cet appel provoque l'exécution d'une portion de code dans le noyau (Kernel). Le **`man 2`** recense plusieurs centaines de ces fonctions système[1].

En fonction de l'endroit où le code était en train de s'exécuter au moment de la réception d'un signal, le comportement du processus n'est pas toujours le même.

Interruption du code en User Space

Si l'interruption a lieu alors que le code s'exécute dans le User Space, une fois le traitement du signal terminé, le processus reprend là où il a été interrompu, comme si rien ne s'était passé.

Ça arrive quand le processus est en train de faire un calcul ou un gros traitement dans une boucle par exemple.

C'est le cas le plus simple car il est sans aucun impact. Il n'y a rien à prendre en compte dans l'écriture du code, c'est totalement transparent pour le codeur.

Interruption du code en Kernel Space

Si l'interruption a lieu alors que le code s'exécute dans le Kernel Space (donc durant un appel à une fonction système), une fois le traitement du signal terminé, le processus **ne reprend pas** là où il a été interrompu, l'exécution reprend dans le User Space en retour de la fonction système qui s'exécutait au moment de l'interruption par un signal.

Cependant, la fonction système ainsi interrompue renvoie une information essentielle pour signifier qu'elle n'a pu mener son travail correctement. Elle renvoie un code **`-1`** qui, comme on l'a appris dans le TP précédent, signifie qu'il y a eu une erreur, et on sait bien que c'est **`errno`** qui va nous aider à y voir clair :

> En cas d'interruption d'une fonction système par un signal, la fonction retourne **`-1`** et **`errno`** vaudra **`EINTR`** qui signifie **`E`**rror **`INT`**e**`R`**ruption

Quand cette erreur **`EINTR`** survient, on a l'assurance que l'action demandée à la fonction système interrompue n'a pas été menée à bien, même pas partiellement. Cette action doit donc être réitérée si besoin, par un nouvel appel à cette fonction système avec les mêmes paramètres. Par exemple, si la fonction était un **`write()`** dans un fichier, on a l'assurance que rien n'a été écrit si le **`write()`** renvoie **`-1`** avec un **`errno`** valant **`EINTR`**.

[1] Un **`ls /usr/share/man/man2`** vous les montrera, c'est là que sont stockés les fichiers sources des manuels du volume 2 (syscalls).

C'est particulièrement pratique de ne pas avoir à se soucier de ce qui aurait pu être partiellement écrit !

Il est à noter que certaines fonctions systèmes utilisent un autre code d'erreur qui est **EAGAIN**. Consultez la section **ERRORS** du manuel de chaque fonction système que vous utilisez et qui est susceptible d'être interrompue.

Voici le code à mettre en œuvre (et à adapter !) quand il y a un risque potentiel qu'un appel système puisse être interrompu par l'arrivée d'un signal :

```
bool sysCallOK = FALSE;

// Un exemple d'appel système
sysCallOK = FALSE;
while (!sysCallOK) {
     if (un_appel_system(...params...) == -1) {
          if (errno == EINTR) {
               continue;  // Interrompu, on recommence (on boucle) !
          } else {
               ... Traiter les autres cas d'erreur ou faire un perror()
               break;  // ou exit ou ce qui est adapté à la situation
          }
     } else {
          sysCallOK = TRUE;  // Ca s'est bien passé, sortie de boucle
     }
}
```

Évidemment, ce type de code n'est à écrire que s'il y a un risque d'interruption par un signal, sinon ce n'est pas nécessaire.

Se pose maintenant la question de savoir quand un tel risque existe. On ne sait pas non plus ce que sont réellement ces signaux qui peuvent survenir, ni ce que signifie "Traiter le signal". C'est l'objet de la suite de ce TP !

1ère expérience

En 1A et 2A, vous avez très certainement déjà expérimenté la réception de signaux, sans forcément le savoir.

Compilez et exécutez le code suivant, sans chercher à le modifier :

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

char *p = NULL;

int main() {
     strcpy(p, "Ca plante !");
     return EXIT_SUCCESS;
}
```

Vous devez obtenir un “Segmentation Fault”, une erreur de segment, un accès interdit à une zone de mémoire qui ne vous appartient pas. En fait, votre processus a reçu un signal **SIGSEGV**. Trouvez-le dans **man 7 signal**.

Au passage, votre œil de lynx aura certainement repéré la portion de code fautive.

Maintenant vous saurez que cette erreur, assez fréquente chez les étudiant.e.s, vous est notifiée grâce à un signal **SIGSEGV**.

A) Emission de signaux

Mise en pause & réveil d’un processus

En SYS 1A, vous avez appris et expérimenté l’usage de **CTRL+Z** pour placer un processus en pause depuis le Bash.

On dit aussi : “stopper” le processus. Attention, c’est bien stopper et non pas arrêter ou supprimer le processus. Le processus reste dans la liste, il est juste sans activité jusqu’à ce qu’on le sorte de cet état. Dans la suite de ce TD, on parlera donc de “le mettre en pause”.

Il existe deux signaux de mise en pause, l’un est réservé au **CTRL+Z** tapé au clavier (**SIGTSTP**[2]) et l’autre (**SIGSTOP**) doit être émis par un appel à la commande **kill** ou la fonction système **kill()**. Les deux ont le même effet (l’un en ligne de commande, l’autre

[2] **T**erminal **ST**O**P**

dans du code en C). Retrouvez dans le manuel (**`man 7 signal`**), les numéros de ces deux signaux.

Compilez le programme suivant pour servir de test :

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

int main() {
     int loop;
     for (loop = 0; loop < 600; loop++) {
          printf("PID %d - Passage %d\n", getpid(), loop);
          sleep(1);
     }
     return 0;
}
```

Notez que la fonction système **`sleep()`** permet simplement d'attendre le nombre de secondes passé en paramètre (ici **`1`** seconde).

Ce programme va vous permettre d'avoir un programme qui égraine les secondes pendant 10 min. Lancez-le dans un Terminal.

Dans un autre Terminal et sans utiliser **`CTRL+Z`** (évidemment !), utilisez la commande adéquate en ligne de commande pour envoyer un signal de mise en pause du programme qui tourne.

Si vous ne trouvez pas la solution par vous-même, demandez de l'aide à ~~Google~~ l'enseignant.e.

Dans le 1er Terminal, observez que la boucle s'est arrêtée. Vérifiez que le processus est toujours présent dans la liste des processus : **`ps -a`**

Dans le manuel (**`man 7 signal`**), trouvez le signal qui permet de sortir un processus de cet état (de le réveiller, de le "dé-stopper" en quelque sorte). Envoyez ce signal à votre processus et observez que la boucle continue là où elle s'était arrêtée.

Mnémoniques

Les signaux sont connus et manipulables par leur numéro ou par leur mnémonique.

On utilise certains signaux avec leur numéro.

Depuis 1A, vous connaissez le **`kill -9 <PID>`** qui permet de mettre un terme à un processus. Le signal en question est numéroté **`9`** (en non pas **`-9`**, car ce n'est pas un "moins" mais un "tiret", comme le **`-a`** dans un **`ls -a`**).

Ce signal **9** possède aussi un nom, qu'on appelle un mnémonique. Tous les mnémoniques des signaux commencent par **SIG**. Par exemple, le signal **9** est **SIGKILL**. Il porte un mnémonique plutôt explicite.

Dans votre code, vous devez impérativement utiliser ces mnémoniques qui sont définis dans **signal.h**.

L'usage impératif des mnémoniques dans du code en C vient du fait que les signaux de **1** à **15** sont standards entre les Unix, mais au-delà de **15**, ces numéros varient. Ainsi, en utilisant des mnémoniques, on s'assure que le compilateur utilisera la bonne valeur associée à un signal donné par son mnémonique.

En ligne de commande, il est conseillé d'utiliser des mnémoniques aussi pour éviter de se tromper. On dira que **9** (**SIGKILL**) et **15** (**SIGTERM**) sont deux exceptions car on est censé bien les connaître. Ainsi on écrira :

```
kill -9 <PID>
```

ou

```
kill -SIGKILL <PID>
```

ou encore, on peut aussi omettre le préfix **SIG** (uniquement en CLI, pas en code C)

```
kill -KILL <PID>
```

A partir de maintenant, utilisez les mnémoniques dans votre code.

Pause par programme

Note : cet exercice fait référence à l'exercice 8 (**argc**, **argv**) du TP 1, qui doit être compris et maîtrisé.

Reprenez l'exemple de l'exercice précédent mais cette fois-ci vous allez créer un programme C, nommé **mon_kill**, qui acceptera comme 1er et unique paramètre, le **PID** d'un processus. Votre programme devra envoyer un signal de mise en pause au **PID** passé en paramètre. Vous devez vérifier la cohérence des arguments du **main(...)** !

Testez votre programme comme pour l'exercice précédent.

Réveil par programme

Créez un programme C, nommé **wake_up**, qui acceptera comme 1er et unique paramètre, le **PID** d'un processus. Votre programme devra envoyer un signal de réveil au **PID** passé en paramètre.

Testez votre programme comme pour l'exercice précédent.

B) Réception de signaux

C'est ici que nous allons parler de ce que signifie "Traiter un signal" et de la façon de procéder.

Fonction signal()

ATTENTION : **signal()** est un faux ami, il ne permet pas d'envoyer un signal, c'est le rôle de la fonction **kill()** de faire cette action ! **signal()** sert à mettre en place une fonction qui sera appelée automatiquement à réception d'un signal donné.

Voici un exemple de code d'utilisation de la fonction **signal()** pour mettre en place une fonction de traitement personnalisé du signal **SIGTERM** qui est le signal qui est envoyé par défaut avec la commande **kill <PID>** :

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

void mon_action(int sig) {
     printf("Recu signal %d\n", sig);
}

int main() {
     int loop;
     printf("%d\n", getpid());
     signal(SIGTERM, mon_action);
     for (loop = 0; loop < 600; loop++) {
          printf("PID %d - Passage %d\n", getpid(), loop);
          sleep(5);
     }
     return 0;
}
```

IMPORTANT : Observez bien que sur la ligne **signal(SIGTERM, mon_action)**, le "**mon_action**" n'a pas de parenthèses. Il s'agit de passer en paramètre l'adresse de la fonction. Avec des parenthèses, on ferait un appel à la fonction et on utiliserait le retour de la fonction comme 2ème paramètre de la fonction **signal()**. C'est quelque chose d'important à bien comprendre.

Allez voir dans le manuel ce que fait **getpid()**.

Compilez et testez ce programme. Cette fois-ci la boucle fait des pauses de 5 sec.

Depuis un autre Terminal, lancez des **`kill`** (sans spécifier de signal) sur le **`PID`** de votre processus.

Lancez plusieurs **`kill`** successifs assez rapidement et observez l'affichage de votre processus. Avez-vous l'impression que les pauses entre 2 affichages font toujours 5 sec quand un signal arrive ? Avez-vous une explication ?

Cas particulier du sleep()

Nous l'avons vu en introduction, quand un signal survient, le programme est interrompu.

Au retour de la fonction de traitement du signal, le programme se poursuit là où il s'était arrêté. Par exemple, si c'est dans une boucle, il poursuit dans la boucle, sur la ligne où il était au moment du signal. Mais s'il était dans un appel d'une fonction système, il retourne immédiatement de cette fonction système.

Ce principe de fonctionnement est valable pour TOUS les appels système.

Cependant, **`sleep()`** n'est pas une fonction système (**`man 3`** et pas **`man 2`**) mais a un comportement similaire aux fonctions système : en fin de traitement du signal, le programme se poursuit comme si le **`sleep()`** était arrivé à son terme, même s'il restait encore un peu de temps de pause (de sleep).

Cherchez, dans le manuel, la nature de ce que **`sleep()`** retourne. Vérifiez cela en affichant la valeur retournée par **`sleep()`** et expérimentez-le en faisant plusieurs **`kill`** rapides successifs, comme à l'exercice précédent. Les valeurs affichées sont-elles cohérentes avec les explications du manuel ?

Fonction sigaction()

La fonction **`sigaction()`** peut être utilisée à l'identique que la fonction **`signal()`**, à savoir, préparer l'appel d'une fonction de traitement qui recevra un unique paramètre : le signal qui l'a déclenchée. Nous n'allons pas étudier ce cas ici, puisqu'il n'apporte pas grande chose de plus que l'usage de **`signal()`**.

Nous allons voir une autre façon d'utiliser **`sigaction()`**, bien plus intéressante.

Cette façon permet de préparer l'appel d'une fonction de traitement qui recevra non pas 1 mais 3 paramètres :

- Le numéro de signal
- Une structure **`siginfo_t`**
- Une variable de contexte, plus rarement utilisée. Nous n'en ferons pas usage non plus.

Exemple :

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

```
void attrape_sig(int sig, siginfo_t *siginfo, void *contexte) {
     // Ici traitement du signal recu
}
```

Le paramètre intéressant est donc le second, la structure **siginfo_t**. Elle contient un nombre important de champs dont :

- **si_pid** : (**int**) le **PID** du processus émetteur du signal
- **si_uid** : (**int**) le **UID** (User ID) du propriétaire du processus émetteur du signal

La mise en place d'une fonction de traitement de signal avec **sigaction()** se fait ainsi :

```
struct sigaction act;

memset (&act, '\0', sizeof(act));
act.sa_sigaction = attrape_sig;
act.sa_flags = SA_SIGINFO;
sigaction(SIGTERM, &act, NULL);
```

A l'aide de ces éléments, re-codez l'exercice précédent (qui utilisait **signal()**), en utilisant **sigaction()** cette fois-ci. La fonction de traitement du signal devra aussi afficher maintenant le **PID** et le **UID** de processus qui a envoyé le signal.

Cas particulier de sigaction() avec SIGTSTP

Reprenez le code de l'exercice précédent pour attraper le signal **SIGTSTP**. C'est le signal **CTRL+Z** tapé au clavier.

Testez votre programme et tapez **CTRL+Z**.

Qu'affiche-t-il comme **PID** et comme **UID** ?

Et en envoyant le signal **SIGTSTP** depuis un autre Terminal, avez-vous un affichage différent ?

Quel usage peut-on faire de cette différence ?

Exercice 1

Ecrivez un programme qui :

1) Attrape un **CTRL+C** (**SIGINT**) effectué au clavier, et affiche le message suivant :

```
Reçu un CTRL+C.
Pour arrêter le programme il faut taper 5 fois CTRL+C.
Il en manque encore 4.
```

2) Attrape un **CTRL+Z** effectué au clavier, et qui affiche le message suivant :

> **Reçu un CTRL+Z.**
> **Remise à zéro du compteur des CTRL+C.**

3) Boucle tant que les **5 CTRL+C** n'ont pas été atteints. La boucle fera des pauses (**sleep**) de 1 seconde pour éviter de consommer du CPU à ne rien faire que boucler.

Rappel : un **CTRL+C** et un **CTRL+Z** provoquent la réception de signaux.

Les 2 signaux doivent être traités par la même fonction.

Votre programme doit quitter à réception du 5ème **CTRL+C**. Le **CTRL+Z** remet le compteur à zéro, comme vous l'aviez compris…

Exercice 2

Il existe 2 signaux sans signification précise, qui peuvent servir pour un usage privé et personnalisé dans nos programmes. Il s'agit de **SIGUSR1** et **SIGUSR2**. On peut en faire les usages qu'on souhaite.

Ecrivez un programme qui doit :

- Faire une pause (sleep) de **60** secondes.
- Quitter après **60** secondes.
- Afficher un message **"Reçu SIGUSR1"**, à réception d'un signal **SIGUSR1**
- Poursuivre sa pause sur le temps restant en cas d'interruption du **sleep()** sur réception d'un signal.

Relisez le paragraphe sur Cas particulier du sleep() pour comprendre la nature de la valeur retournée par **sleep()**.

Affichez des messages aux endroits utiles pour suivre le déroulement du programme.

Testez votre programme en lui envoyant des signaux **SIGUSR1** depuis un autre Terminal.

Le retour du SIGSEGV !

Maintenant que vous savez attraper un signal, essayez de mettre en place un traitement du **SIGSEGV** rencontré à l'exercice **1ère expérience** en début de TP.

Quelle solution pourrait-on proposer, dans la fonction de traitement du signal, pour corriger l'erreur ? Est-il possible d'allouer de la mémoire dans cette fonction pour que **p** puisse pointer vers une vraie zone de mémoire (non **NULL**) ? Est-ce que vous parvenez à quelque chose de stable ?

Note (à lire une fois cet exercice terminé) : ceci est un exercice qui n'a d'intérêt que pour les besoins de ce TP. Dans la vraie vie, on ne tentera pas ce genre de traitement sur **SIGSEGV** car une telle erreur dans le code ne pourra que mener à des bugs et à une

instabilité plus loin dans le code. Si un pointeur ne pointe pas vers une zone de mémoire correcte, il y a toutes les chances que la suite du code ne soit pas exploitable. D'ailleurs vous avez sans doute eu des difficultés à résoudre ce problème, si vous y êtes arrivé !

Bouquet final, juste pour rigoler...

Attention, ne faites pas ce qui suit si vous avez des fichiers ouverts et non sauvés.

Quand c'est OK, vous pouvez ouvrir plusieurs Terminaux, lancer des Navigateurs, ouvrir des PDF, etc. et tester ce qui suit.

La commande **`kill -<SIG> -1`** (le chiffre **`1`**, pas un **`L`** minuscule) permet d'envoyer le signal **`SIG`** à tous les processus sur lesquels vous avez des droits. Donc, pour résumer, tous vos processus personnels, les Terminaux, Navigateurs, etc. que vous avez lancés.

Expérimentez alors : lancez un **`kill -KILL -1`**

Au revoir...