

TP 1 – Les signaux



⚠ Mission pour réussir sans soucis la traversée du *Système 2*

- 👉 Il y a un pont suspendu entre l'exercice 2 et la suite des TPs en Système 2 : c'est le seul passage ! Un détour serait trop coûteux voire dangereux !
- 👉 Cependant le passage est facile si le Fork, Excel, Wait sont bien fixés.
- 👉 Faites tout d'abord les exercices 1 et 2 sur les signaux.
- 👉 Au moment de passer le pont, 3 solutions s'offrent à vous :
 - (a) Vous pensez passer sans soucis avec votre expérience en Système 1 : Tester rapidement le fork, l'excel et le wait avec l'exercice 3 proposé. Ça fonctionne ? Go.
 - (b) Votre expérience vous paraît loin : Tester rapidement le fork, l'excel et le wait avec l'exercice 3 proposé. Le pont tangue et fait des bruits de craquements, c'est trop risqué pour la suite ? Pas de panique vous avez 3 semaines pour consolider les problèmes.
 - (c) Aucune expérience dans la construction système avec fork, execl, wait ? Pas de panique c'est relativement facile. Faites l'exercice 3 proposé et jetez un oeil sur les pointeurs indiqués. Vous avez 3 semaines pour passer fièrement le pont.

1 Interception

- ▶ Réalisez un programme qui incrémente (et affichera) un compteur de 0 à 60 toutes les 500 ms (utilisez la fonction *usleep*). Dans ce programme, réalisez l'interception du signal CTRL+C (SIGINT) : le programme affichera un message avant de quitter. **Utilisez l'exemple du cours pour la mise en place du *handler*.**

⚠ Remarques

- 👉 la fonction *usleep* prend en argument des microsecondes ($1\text{ s} = 1000\text{ ms} = 1000000\mu\text{s}$).
- 👉 $500\text{ ms} = \dots \mu\text{s}$

**aide C**

- ✎ compilation : `gcc prog.c -o prog` (prog est l'exécutable).
- ✎ Pour écrire les messages d'erreurs, vous utiliserez la fonction `perror("...")` suivie d'un `exit(n)` où n est différent de 0.
- ✎ Pour les autres messages, vous utiliserez la fonction `printf("...")`; suivie de `fflush(stdout)`; pour s'assurer de l'écriture à l'écran.

- Ensuite, faites en sorte que le programme ne puisse être quitté qu'au deuxième CTRL+C. Le programme affichera un message après le premier et le deuxième CTRL+C.

**Remarques**

- ✎ Malheureusement comme la fonction handler ne prend pas de paramètre (sauf le numéro du signal reçu), le seul moyen de stocker de l'information est d'utiliser des variables globales ou des variables *static*.

- Enfin, scindez votre boucle en 2, avec une de 0 à 40 et la seconde de 40 à 60. Autour de la première boucle mettez en place un masque pour le signal SIGINT, puis enlevez-le après (c'est-à-dire qu'il ne faut plus de masque durant la deuxième boucle). **Utilisez l'exemple du cours pour la pose du masque.**

Pendant l'exécution du programme et avant 40, envoyez 2 CTRL+C. Observez le comportement du programme. Pourquoi n'a-t-il pris en compte qu'un seul des deux signaux à la sortie de la zone masquée ?

Réponse : Les signaux "pendants" (=envoyés et en attente d'être délivrés) sont stockés dans un tableau binaire : 0 ou 1, donc on ne peut pas mémoriser l'arrivée de plusieurs signaux de même type tant que le premier n'est pas délivré et que le bit n'est remis à 0.

2 Timer

L'objectif est de réaliser un jeu qui lance un timer et dont le but est d'envoyer un CTRL+C le plus proche de la fin du timer. [lisez tout ce qui suit avant de commencer à coder]

- Vous utiliserez la fonction `alarm(n)`, qui déclenche un timer de n secondes : au bout de ces n secondes, un signal SIGALRM est envoyé au processus. Nous prendrons `n=10`.
- Si le CTRL+C arrive avant la fin du timer, le programme affichera le temps restant sinon il affichera un message de type "perdu".

Pour récupérer la valeur du temps restant, 2 possibilités :

- ☞ `alarm(0)` désactive le timer et renvoie le nombre de secondes qu'il restait avant la réception du signal (de l'alarme qui était en cours).
- ☞ ou utiliser la fonction `time(NULL)` pour en déduire le temps restant (en faisant la différence entre l'instant de départ et l'instant de réception).

- ▶ Le programme pourra attendre avec la fonction `pause()`. (les opérations d'affichage se feront dans le(s) handler(s))
- ▶ Il est possible de réaliser un seul handler pour les deux signaux (en utilisant l'argument du handler). Mais ce n'est pas obligatoire et peut-être même moins lisible.

3 On arrive au pont !



Faites l'exercice et voyez si vous êtes plutôt (a), (b) ou (c)...(voir entête de l'énoncé)

Utilisation de *fork* et *execl*

Prenez l'habitude de récupérer les exemples du cours pour ne pas perdre de temps et éviter de réinventer la roue ou de recopier des choses fausses provenant du web...

a. Création simple

Écrire un programme C créant un processus père et un processus fils. On prendra soin de réaliser des affichages à l'écran pour savoir où l'on se situe lors de l'exécution. Vous afficherez les pid de chaque processus (`getpid()` et `getppid()`).

Prenez l'exemple donné en cours et adaptez le.

b. Recouvrement avec *execl*

On veut appeler un programme avec 3 arguments. Ces 3 arguments seront transmis à un autre programme qui fera une opération dessus.

Ainsi en tapant **Pere 8 6 trucs** on doit obtenir "J'ai 48 trucs". L'affichage et la multiplication sera faite en réalité par une autre programme (dit recouvrant).

Écrire deux programmes C :

- ▶ Le premier est le processus père : Il crée le processus fils (avec `fork` comme le précédent). Le programme père prendra 3 arguments (2 entiers et une chaîne de caractères). Ces

arguments (contenus généralement dans `argv` dans la fonction `main`, voir l'encart ci-dessous) sont des chaînes de caractères (même les entiers). Mais comme le processus père ne manipulera pas ces arguments, il ne fera que les transmettre à son fils, il n'aura donc pas besoin de les convertir. La transmission est d'ailleurs transparente car après le `fork`, le fils connaît aussi directement les arguments du père (`argv`) puisqu'il y a recopie des variables. Avant de quitter le père devra attendre la fin de son fils (avec `wait` ou `waitpid`)

- Le deuxième est le programme qui recouvrira le programme du fils. Ce programme prendra 3 arguments (2 entiers et une chaîne de caractères) et il affichera la chaîne de caractères et le produit des deux entiers. Les trois arguments seront transmis "par le père" par l'intermédiaire de la fonction `execl` dans la partie fils du code (après le `fork`).

Exemple : **recouvrant 8 6 trucs** affichera par exemple "J'ai 48 trucs". Ainsi **Pere 8 6 trucs** lancera un fils qui exécutera **recouvrant 8 6 trucs**.

N'oubliez pas de compiler les 2 programmes (séparément). Vous pouvez là encore afficher des messages pour bien comprendre ce qui se passe et **vous devez mettre un message d'erreur juste après la fonction `execl(...)` (message qui sera affiché seulement si le `execl` ne fonctionne pas) ainsi que `exit` pour éviter que le programme ne continue à s'exécuter.**

Prenez l'exemple donné en cours (pour voir où se situe `execl`) et adaptez-le.

aide C

- ☞ La fonction `main` d'un programme prend comme premier paramètre le nombre d'arguments (`int argc`) de l'exécutable et stocke ces arguments dans le deuxième paramètre qui est un tableau de chaîne(s) de caractères (`char * argv[]`) : `argv[0]` est la chaîne de caractère contenant le nom de l'exécutable, `argv[1]` le premier argument, ...
- ☞ Pour les paramètres de la fonction `execl(...)`, il faut donc mettre après le premier paramètre (qui est le chemin de l'exécutable), tous les arguments (de 0 à N) puis finir par `NULL`. Exemple : `execl("./recouvrant", "recouvrant", argument1, argument2, NULL)` ; avec `argument1` et `argument2` des chaînes de caractères (par exemple `char argument1[10]` ;).
- ☞ Pour transformer une **chaîne en entier** (vous en aurez besoin dans le programme `recouvrant`, passer "8" en 8), on peut utiliser `sscanf(chaine, "%d", &entier)` ou la fonction `atoi(chaine)` (`AsciiToInt`) qui renvoie l'entier.
- ☞ Pour transformer un **entier en chaîne de caractères** (dans cet exercice vous n'en aurez pas besoin), vous pourriez utiliser la fonction `sprintf(chaine, "%d", 8)` ; (8 étant l'entier)

Important

Ce module Système 2 implique que vous possédiez les notions abordées en Système 1 (*fork*, *wait*, ...), et ce, que vous proveniez de S4 de cette Licence ou d'un autre parcours (IUT, ...).
Avant le TP2, refaites/parcourez les TP1 et TP2 du Semestre S4 (tout est indiqué sur le cours ARCHE de Système 2).



Si la traversée a été tranquille, vous pouvez passer à l'exercice suivant.

4 Timer++

Cette fois-ci, ce seront deux processus fils qui s'affronteront dans le maintenant-devenu-célèbre jeu du timer.

- ▶ Le père créera 2 fils, puis le père enclenchera le timer. Vous utiliserez *execl* pour les fils (on pourra dissocier les fils en envoyant 1 ou 2 par argument). [il a donc 2 programmes .c : un pour le père, un pour les fils]
- ▶ Le premier fils devra envoyer le signal SIGUSR1 et le deuxième le signal SIGUSR2 au père. Pour simuler la compétition : l'envoi devra se faire aléatoirement entre 5 et 12 s en utilisant *usleep(n)* (*n* en μ s) pour faire attendre le fils (ce qui veut dire que de temps en temps le signal sera envoyé après la fin du timer...).

Aide en C

 Pour créer un générateur aléatoire de nombre :


```
#include <stdlib.h>
#include <time.h>
/* Initialisation du générateur au début de chaque fils avec par exemple :*/
srand(getpid()); //ici srand(time(NULL)) n'est pas intéressant
//car les fils sont créés quasiment en même temps
//donc le nombre aléatoire sera identique !

/* génération d'un nombre */
int i = rand(); // nombre entre 0 et RAND_MAX (exclus)
float f = rand()/(float)RAND_MAX; // ici entre 0 et 1. (exclus)
int j = rand()%N; // ici entre 0 et N (exclus)
```

- ▶ Le père doit afficher le gagnant quand il aura reçu les deux signaux ou que le temps soit dépassé, les temps de réception seront stockés dans des variables, puis comparés à la fin du timer dans le handler de SIGALRM. Faites déjà tourner votre programme avant de passer à la question suivante. L'attente du père devra se faire avec des `pause()` pas des `Wait` (`sigalrm` désactive automatiquement le `wait` ou `waitpid`!)
- ▶ Question : que risque-t-il de se passer si un des fils déclenche le handler pendant l'exécution d'un autre handler ? Que faire pour y remédier ?

Réponse : Le handler risque d'être interrompu au mauvais moment (après le test du nombre de signaux reçus ou la mise à jour d'une variable). Les résultats peuvent devenir incohérents. Solution : mettre un masque sur les handlers qui gèrent les SIGALRM/SIGUSR1/SIGUSR2 pour masquer les signaux (dans le `sigaction`) : vous pouvez faire un même masque avec les 3 signaux que vous utiliserez pour les 3 `sigaction`, même si par exemple masquer SIGALRM pour `sigaction` de SIGALRM est redondant (puisqu'un masque par défaut).

Aide en C

 Pour mettre un masque "autour" d'un handler, il faut placer le masque au moment de l'initialisation de la structure `sigaction` et il sera effectivement mis en place lors de la fonction `sigaction`, puis enlevé à la sortie.