

## Atelier n°3 : Processus et signaux.

### Création de processus : introduction.

Un **processus** correspond à un **programme en cours d'exécution**. Il s'agit d'une entité **active** et **dynamique**, à l'inverse d'un programme ou d'un script, entités « passives », qui décrivent des opérations sans pour autant les réaliser. La confusion entre « programme » et « processus » est courante, mais on essaiera à partir de maintenant de l'éviter.

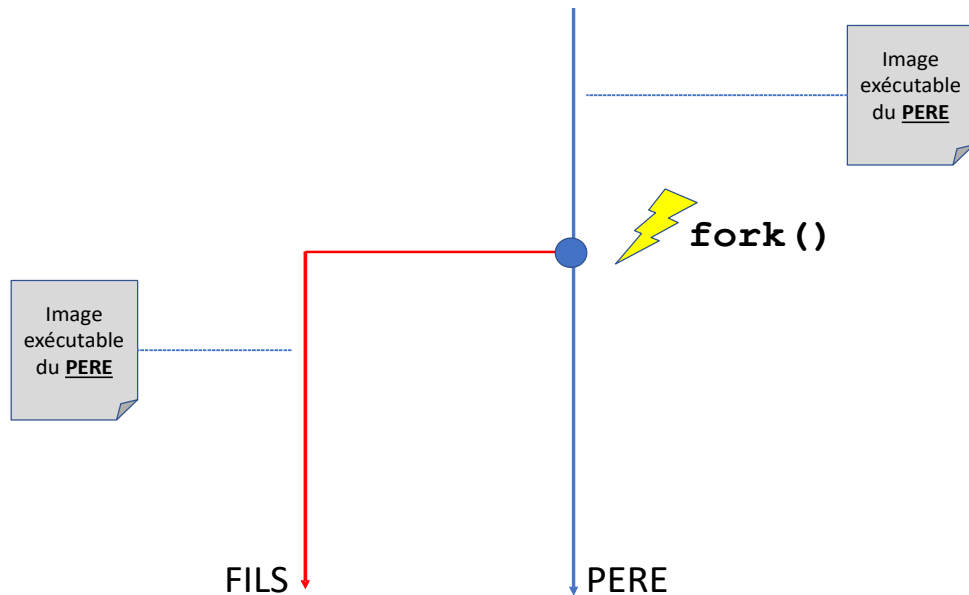
Pour les besoins de son activité, un processus a besoin de **ressources**, essentiellement pour **communiquer** et **calculer**. En définitive, celles-ci peuvent se résumer à :

- des **entrées** et des **sorties** (canaux de communication, par exemple ceux gérés par l'intermédiaire de `stdin`, `stdout` et `stderr`, des « *sockets* » réseaux, des canaux ouverts sur des fichiers, *etc.*).
- de la **mémoire** (pour stocker les données et pouvoir travailler dessus : un processeur nécessite que les informations requises par les traitements résident, ne serait-ce que transitoirement, dans certains de ses « registres » ou dans des composants de mémoire).
- Du « **temps processeur** », qui correspond au temps que peut consacrer un processeur ou un cœur de processeur au traitement du processus. Ce temps processeur est *a priori* découpé en « *time slices* » qui sont attribuées au processus « élu » par l'ordonnanceur. Cette attribution est remise en cause et ré-évaluée en fonction des besoins des autres processus (dits « concurrents »), suite à des événements particulier (disponibilité ou absence de données, délai, *etc.*) ou de manière périodique. Ce mécanisme de **temps partagé** permet un **parallélisme d'exécution** sur une échelle de temps « *macroscopique* » (c'est à dire si on observe l'activité du système pendant une durée correspondant à un nombre suffisamment important de « *time slices* »), même sur des systèmes « mono-cœur » (en gardant à l'esprit qu'à un instant  $t$  donné, une seule tâche s'exécute). Sur les systèmes multi-cœurs, le parallélisme est à la fois « physique » (il y a plusieurs activité en parallèle à un instant précis) et issu de ce mécanisme (« *soft* »).

On retiendra par conséquent que la création d'un processus induit l'attribution de ressources suffisante pour son activité. Celles-ci ne sont pas illimitées.

La fonction de base sous Linux ou Unix pour créer un processus est « l'appel système » `fork()`. Celui-ci a un fonctionnement un peu « étrange » de prime abord. En effet, suite à l'appel à `fork()` par le processus « créateur » (qu'on nommera processus « père »), le système d'exploitation **crée une copie conforme de ce processus père**, c'est à dire un nouveau processus **exécutant le même code** que son géniteur et utilisant un **jeu de données qui est une copie de celles du père** (**attention**, il s'agit bien d'une copie : modifier les variables du père n'aura pas d'incidence sur celles du fils et *vice-versa*).

Le schéma ci-dessous précise la chronologie des événements (la flèche du temps est orientée vers le bas) : au moment de l'appel à `fork()`, il y a création du fils qui exécute le même code que le père :



Le programme ci-dessous est un premier exemple d'utilisation de cet appel système. On note `test-fork.c` ce fichier source. Il se compile de manière simple *via* `gcc`, par exemple par :

```
$ gcc test-fork.c -o ./test-fork
```

Il faut lancer ce programme et tenter de répondre aux questions suivantes :

- comment sait-on si l'appel à `fork()` a échoué ?
- A quoi sert la variable `errno` ?
- A quoi sert l'appel `strerror(errno)` ?
- Comment sait-on que le code s'exécute dans le « contexte » du père ?
- Comment sait-on qu'il s'exécute dans le « contexte » du processus fils ?
- A quoi correspondent les valeurs retournées par les appels `getpid()` et `getppid()` ?
- Quel(s) intérêt(s) de telles valeurs présentent-elles ?
- Comme indiqué dans le fichier source, faire un second essai en commentant la ligne comportant l'appel `id = wait(...)`. L'affichage produit est-il cohérent ? En déduire le rôle de la fonction `wait()`.
- Question *a priori* étrange : existe-t-il un « processus au grand coeur » prenant sous son aile tous les orphelins (*indications* : la commande `$ ps` aux permet de lister tous les processus en cours d'exécution sur la machine. Les numéros de processus sont attribués par ordre croissant depuis le démarrage du système) ?

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>          /* ->recuperation des codes d'erreur */
#include <sys/types.h>
#include <sys/wait.h>
/* ===== */
/* programme principal */
/* ===== */
int main( void )
{
    pid_t    id_fils,        /* ->Process ID du processus fils */
            id;              /* ->ID du processus termine */
```

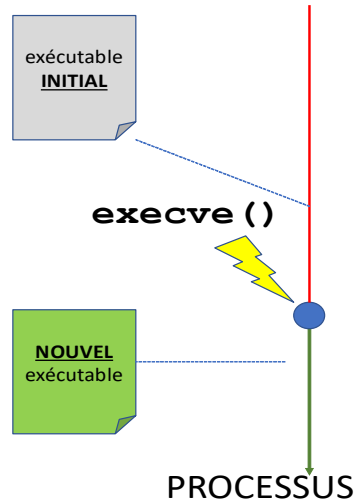
```
char      rep;                /* ->saisie utilisateur                */
int       status;             /* ->"exit status" du fils                */
/* lancement du processus fils */
id_fils = fork();             /* ->creation d'une copie du processus en cours */
/* est-ce que ca a marche ? */
if( (int)(id_fils) < 0 )
{
    /* ECHEC */
    fprintf(stderr,"ERREUR : main() ---> appel a fork().\n");
    fprintf(stderr," code d'erreur = %d (%s)\n", errno,(char *) (strerror( errno)) );
    exit( errno );
};
/* fils ? */
if( (int)(id_fils) == 0 )
{
    printf("FILS:je suis le processus fils de PID %d\n", getpid());
    printf("FILS:mon pere a le PID %d\n", getppid());
    printf("FILS:saisir un caractere + [ENTREE] pour finir...\n");
    fflush( stdin );
    scanf("%c", &rep );
    return( 0 );
};
/* pere ? */
if( (int)(id_fils) > 0 )
{
    printf("PERE: je suis le processus pere de PID %d\n",getpid());
    printf("PERE: mon fils a le PID %d\n", (int)(id_fils));
    printf("PERE: mon pere a le PID %d\n", getppid());
    /*.....*/
    /* FAIRE UN ESSAI EN SUPPRIMANT CETTE LIGNE (id = wait(..) */
    /* !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!*/
    id = wait( &status );
    printf("PERE: Le processus fils %d s'est termine.\n", (int)(id));
    printf("PERE: avec le status de sortie = %d\n", status );
};
/* fin */
return( 0 );
}
```

✍ A votre avis, quelle est la limitation fondamentale de l'appel système `fork()` ?

Linux propose un second mécanisme qui peut permettre de lever cette limitation : il s'agit du « **recouvrement mémoire** ».

### Une primitive de recouvrement.

Le recouvrement mémoire consiste, en cours d'exécution d'un processus, **à venir remplacer à la fois le code exécuté et l'ensemble du jeu de données** par leurs contreparties respectives issues d'un fichier exécutable. Il est important de bien noter que lors de cette opération, **il n'y a pas création d'un nouveau processus**, mais substitution (on parle alors de « recouvrement ») des zones de programme et de données d'un processus existant par celles issues du fichier exécutable dont le chemin d'accès et passé en paramètre à la fonction réalisant cette opération, en l'occurrence ici `execve()` . Le schéma ci-dessous résume son effet :



Pour tester cet appel, nous allons tenter de « recouvrir » le code exécutable initial par celui correspondant au programme dont le fichier source (`sinus.c`) est le suivant :

```

/*=====*/
/* exemple d'un programme simple a lancer */
/* (affiche les valeurs d'une sinusoide) */
/*=====*/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
/* constante */
#define ANGULAR_STEP 0.01
#define WIDTH 2.0
/*=====*/
/* calcul de la suite de valeurs */
/*=====*/
int main( void )
{
    int i;          /* ->compteur de boucles      */
    int iMaxIter;   /* ->borne d'iteration      */
    double dbAngle; /* ->valeur angulaire courante */
    /* initialisation */
    if( ANGULAR_STEP != 0.0)
    {
        iMaxIter = (int)(2.0 * WIDTH / ANGULAR_STEP);
    }
    else
    {
        return( 0 );
    }
    /* calcul effectif */
    for( i=0, dbAngle = 0.0; i < iMaxIter; i++, dbAngle+= ANGULAR_STEP)
    {
        printf("sin(%lf) = %lf\n", dbAngle, sin(dbAngle) );
    };
    return( 0 );
}

```

Le programme initial sera quant à lui le suivant (on nommera `test-execve.c` le fichier source correspondant et `test-execve` l'exécutable généré suite à sa compilation):

```

#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

```

```

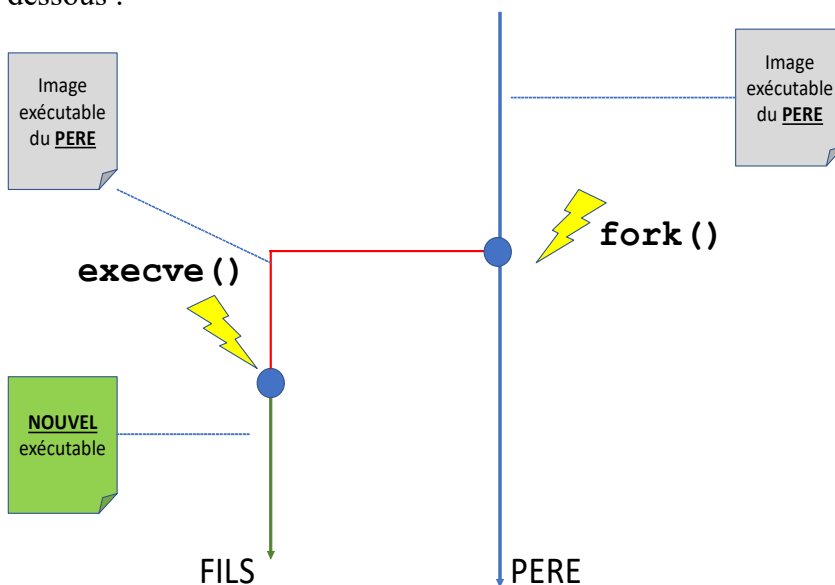
#include <errno.h>
#include <sys/types.h>
/* constantes */
#define NEW_PGM_NAME    "../bin/sinus"    /* ->adapter ce chemin a VOTRE environnement */
#define MAX_LEN        256              /* ->longueur maximale pour les chaines saisies */
/* ===== */
/* programme principal : on attend la saisie d'une */
/* chaine de caracteres pour effectuer le recouvrement */
/* ===== */
int main( int argc, char *argv[] )
{
    char szInString[MAX_LEN];
    printf("saisir une chaine pour faire l'appel execve()");
    scanf("%s", szInString);
    execve(NEW_PGM_NAME, argv, NULL);
    printf("ce message pourra-t-il s'afficher ???\n");
    return( 0 );
}

```

Tester le fonctionnement de `test-execve`. Il faudra prendre soin d'affecter à la constante `NEW_PGM_NAME` la chaîne correspondant au chemin d'accès au programme `sinus` « vu » du répertoire d'exécution de `test-execve`.

### Créer un nouveau processus exécutant une image différente de celle du père.

Pour créer un processus exécutant une image différente de celle du père, la solution consiste à combiner les appels `fork()` et `execve()` suivant la chronologie représentée sur l'illustration ci-dessous :



En vous inspirant de ce qui précède et sur la base des sources des programmes `test-fork` et `test-execve`, écrire le programme **cree-process** qui devra :

- créer un processus fils exécutant l'image **sinus** dès que l'utilisateur aura saisi une chaîne quelconque.
- Attendre la fin de l'exécution du processus fils et afficher « **FIN DU PROCESSUS FILS** » une fois celui-ci terminé.

### Les signaux.

Une part importante des mécanismes de synchronisation entre les processus et le système d'exploitation et, par extension, entre les processus eux-mêmes, repose sur un mécanisme de communication simple et efficace portant le nom de « **signal** ».

Du point de vue du processus, la réception d'un signal peut conduire (dans la mesure où certaines conditions sont satisfaites, voir plus bas) à un comportement analogue à celui qu'on peut avoir lorsqu'on reçoit un coup de téléphone.

En effet, on peut s'imaginer être occupé par une activité quelconque. A ce moment, le téléphone sonne. Compte-tenu de la priorité que nous accordons à notre activité présente, nous pouvons choisir :

- (a) de répondre immédiatement,
- (b) de ne pas répondre, mais de rappeler plus tard le correspondant, lorsqu'aucune activité plus prioritaire ne nous accapara. Dans le contexte des signaux Linux, on dirait alors qu'on « masque » le signal.

Supposons que l'alternative (a) soit celle que nous choisissons. Nous répondons à l'appel puis, une fois celui-ci terminé, nous reprenons l'activité que nous avons dû quitter momentanément.

Ceci à l'esprit, voici comment les choses se passent schématiquement pour les signaux Linux :

- un processus peut demander au système d'exploitation d'envoyer un « signal » (qui revêt la forme d'un code entier) à un processus « cible ». Cette cible peut être le processus émetteur lui-même (voir les « **alarmes** » et les « **alarmes cyclique** »).
- le système émet le signal à destination du processus cible.
- Si le signal **peut être « dérouté »** (c'est à dire s'il est possible de changer le comportement par défaut à la réception du signal) **ET** si la « cible » a configuré une fonction particulière (qu'on appelle **routine d'interception** ou « *signal handler* ») à appeler à réception de celui-ci **ET** si le signal n'est pas « masqué » **ET** si la cible ne traite pas déjà un autre signal alors cette fonction est appelée « immédiatement ».
- Une fois le traitement réalisé par la routine d'interception, le processus reprend le cours de son exécution juste après l'endroit où il avait été interrompu, sauf s'il existe des signaux en fil d'attente qu'il n'avait pas pu traiter car déjà occupé à en gérer un (appels en attente dans le cas du téléphone, « *signal pending* » dans le cas des signaux Linux).

Il faudra prendre garde aux faits suivants :

- certains signaux ne sont pas « déroutables », c'est à dire qu'il n'est pas possible d'y associer une routine d'interception pour en modifier le comportement par défaut.
- Si un processus reçoit un signal « déroutable » pour lequel il n'a pas installé de routine d'interception, alors il est détruit !
- Les appels systèmes associés aux entrées – sorties (notamment la fonction `read()`) sont interrompus par la réception d'un signal alors qu'ils sont par défaut « bloquant ».

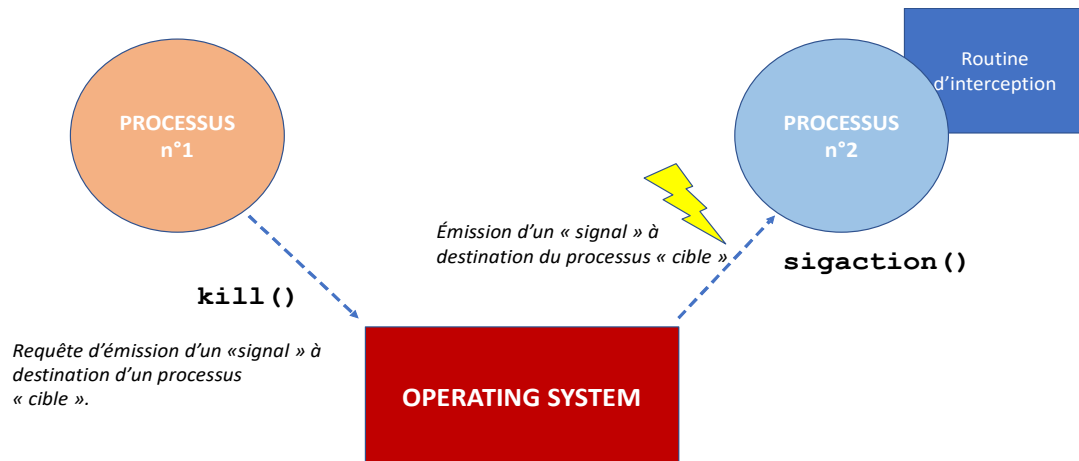
L'appel permettant d'émettre un signal à destination d'un processus est **kill()**. Il existe aussi en ligne de commande (et s'appelle de même `kill`). Par exemple, la commande :

**\$ kill -l**

affiche la liste des signaux gérés par le système, en faisant apparaître leur nom symbolique et le code entier correspondant. Si on souhaite, depuis la ligne de commande, émettre un signal à destination d'un processus cible, le synopsis de `kill` est alors :

`$ kill -s <nom du signal> <PID du processus cible>`

Le schéma ci-dessous synthétise une partie de ces notions :



L'installation d'une routine d'interception repose sur la fonction **sigaction()** et sur la structure **struct sigaction** (malheureusement, la fonction et la structure portent le même nom...). La structure `struct sigaction` permet de paramétrer la manière dont le signal sera « intercepté ». On y précise un « *flag* » qui va contraindre la « signature » (type de donnée renvoyé et type des données passées en paramètre) de la routine d'interception, la liste des signaux à éventuellement masquer et (surtout) le nom de la fonction qui va être appelée à la réception du signal. Avec la donnée membre **sa\_flag** de la structure `struct sigaction` positionnée à 0, la signature de cette routine d'interception est nécessairement **void nom\_de\_la\_routine( int )**. La valeur entière passée en paramètre correspond au code du signal reçu.

Le fichier source ci-dessous est un exemple simple de programme capable de recevoir des signaux. On nommera `test-signal.c` le fichier source correspondant.

🔧 Compiler `test-signal.c` pour générer l'exécutable `test-signal`.

🔧 Vérifier son fonctionnement en le lançant dans un terminal puis, dans un autre, en utilisant la commande `kill` (indication : l'utilisation de la commande `ps` pourra s'avérer utile).


```
/*=====*/
/* programme de test de la notion de signal sous Linux */
/*-----*/
/* Jacques BOONAERT - cours AMSE */
/*-----*/
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <signal.h>                                /* ->GESTION DES SIGNAUX */

extern char *strsignal( int );
/*.....*/
/* prototypes */
/*.....*/
void signal_handler( int );                        /* ->routine de gestion du signal recu */
/*~~~~~*/
/* routine de gestion du signal recu */
/*~~~~~*/
void signal_handler( int signal )                  /* ->code du signal recu */
{
    printf("signal recu = %d\n", signal );
    /* strsignal retourne la chaine de caracteres */
    /* qui correspond au "nom symbolique" du signal */
    printf("%s\n", (char *)strsignal( signal ) );
}
/*~~~~~*/
/* programme principal */
/*~~~~~*/
```

```
int main( int argc, char *argv[] )
{
    struct sigaction    sa;          /* ->structure permettant de definir le gestionnaire */
                                   /* et d'y associer le signal a traiter, etc. */
    sigset_t            blocked;     /* ->contiendra la liste des signaux qui seront masques */
    /* on ne bloque aucun signal : blocked = vide */
    sigemptyset( &blocked );
    /* mise a jour de la structure sigaction */
    memset( &sa, 0, sizeof(struct sigaction));
    sa.sa_handler = signal_handler; /* ->precise le gestionnaire a utiliser */
    sa.sa_flags   = 0;              /* ->fonctionnement "normal" */
    sa.sa_mask    = blocked;        /* ->indique les signaux masques */
    /* installation EFFECTIVE du gestionnaire */
    sigaction( SIGUSR1, &sa, NULL ); /* ->associe signal_handler a la reception de SIGUSR1 */
    sigaction( SIGUSR2, &sa, NULL ); /* ->idem pour SIGUSR2 */
    /* attente de reception de signal */
    do
    {
        printf("attente de signal...\n");
        pause(); /* ->on ne fait rien, hormis attendre un eventuel signal */
    }
    while( 1 ); /* !!! boucle infinie */
    return( 0 );
}
```

### Alarme.

On peut concevoir une alarme comme un mécanisme particulier d'émission de signal. En effet, dans ce cas, le processus demande au système de lui envoyer à lui-même un signal (de code `SIGALRM`) après une durée spécifiée en secondes. Pour effectuer cette requête, le processus appelant utilise la fonction `alarm()` en lui passant en paramètre la durée voulue (par exemple `alarm(10)` enverra au processus appelant le signal `SIGALRM` 10 seconde après l'appel). Si on passe la valeur 0 à `alarm()`, alors l'alarme est désarmée (un peu comme si on avait désamorcé une bombe à retardement).

 A des fins d'illustration, compléter le code ci-dessous pour mettre en place un mécanisme de « *timeout* » sur la lecture de données clavier (on utilise le fait que, comme précisé plus haut, la réception d'un signal interrompt les appels système, tels que `read()`) :

```
/*-----*/
/* programme de test d'une alarme */
/*-----*/
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <ctype.h>
#include <sys/types.h>
#include <signal.h>
/* constantes */
#define DELAI          /* ->temps dont on dispose pour appuyer sur une touche + [ENTREE] */
/* globales */
int GoOn = 1;          /* ->contrôle d'exécution de la boucle */
/* declaration */
void SignalHandler(int ); /* ->routine d'interception */
/* routine d'interception */
void SignalHandler( int signal )
{
    if( signal == SIGALRM)
    {
        printf("TROP TARD...\n");
        GoOn = 0;
    };
}
/*#####*/
```



```

/* mainline */
/******/
int main( int argc, char *argv[] )
{
    struct      sigaction sa;
    struct      sigaction sa_old;
    sigset_t    empty;
    char        cRep;          /* ->pour lecture d'un caractere */
    /* installation de la routine d'interception */
    memset( &sa, 0, sizeof(struct sigaction));
    memset( &sa_old, 0, sizeof(struct sigaction));
    sigemptyset( &empty);
    sa.sa_handler = SignalHandler;
    sa.sa_mask = empty;
    sa.sa_flags = 0;
    sigaction( SIGALRM, &sa, &sa_old);

    do
    {
        /* (1) on arme l'alarme */
        /* TODO... */
        printf("vous avez %d seconde pour entrer un caractere...\n", DELAI);
        alarm( DELAI);
        /* (2) on attend l'appui sur une touche (cf tableau ) */
        /* TODO ... */
        /* (3) si on a saisi un caractere a temps, on desarme */
        /* l'alarme (alarm(0)), sinon on quitte a boucle et le */
        /* programme */
        /* TODO... */
    }
    while( GoOn);
    printf("FIN\n");
}

```