# TP 1.1 (3h): Gestion des processus

**Environnement de travail :**

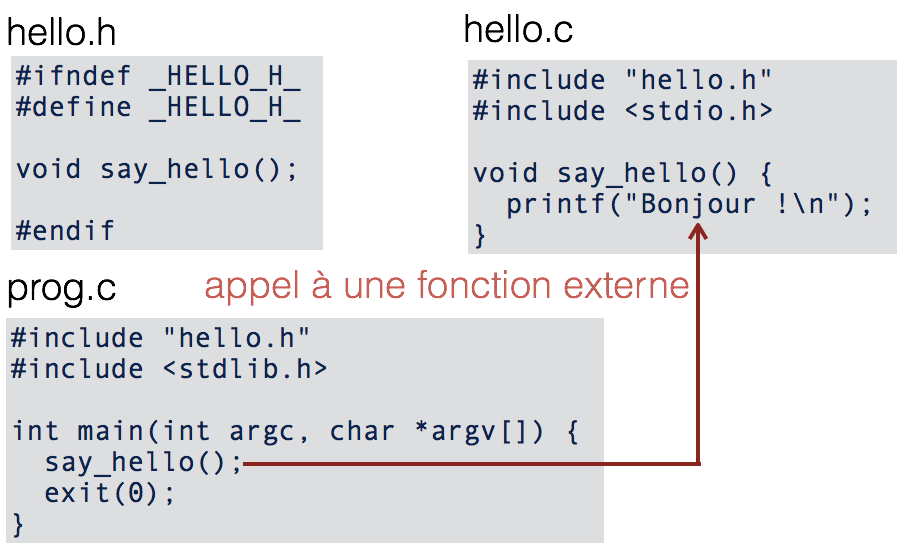
* VM Ubuntu
* deux choix pour éditer/compiler vos programmes C
  + édition avec l’éditeur préféré (Sublime, Scite…)  
    compilation en ligne de commande avec **gcc**   
     exemple : gcc -o nom\_executable fic1.c fic2.c …
  + utilisation de l’IDE CodeBlocks : cela permet d’utiliser le débogueur, et c’est pas du luxe !  
    A noter : *créer un projet séparé CodeBlocks pour chaque exercice de TP !*

**Avant de commencer**, rappelez-vous les fondamentaux du *shell* UNIX et les principales commandes, en particulier la commande **man**. A partir de ce point, lorsqu’une commande ou fonction système est mentionnée, il est conseillé de consulter son manuel avec **man.**

**Documentation de l’API système :** comme pour les commandes shell, le manuel UNIX (**man**) contient des entrées également pour les fonctions C de l’API système. Attention, *parfois le nom d’un fonction C coïncide avec le nom d’une commande shell* ou d’une autre entité listée dans le **man** (c’est le cas par exemple pour **open**).Dans ce cas, il est utile de savoir que le man est *organisé en* *sections* nommées/numérotées, et qu’on peut demander une page de manuel à partir d’une section en particulier. Essayez les commandes suivantes et comparez les affichages :

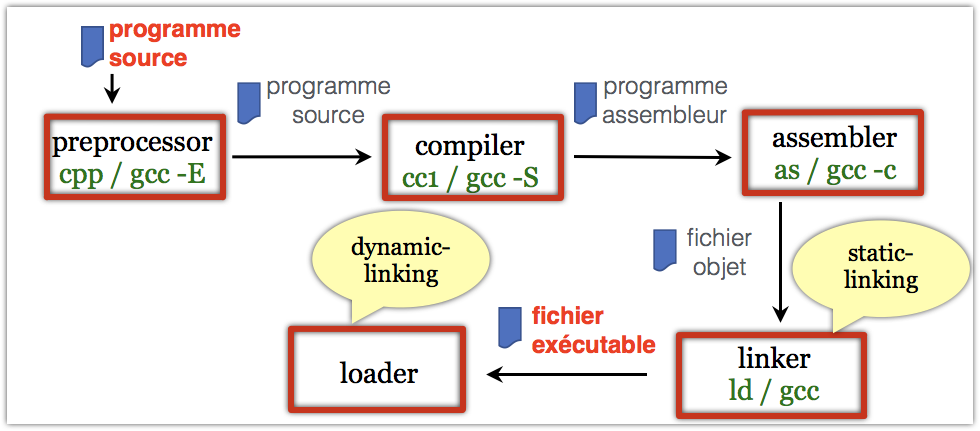
* **man open** (ouvre la première page intitulée *open* trouvée)
* **man 2 open** (ouvre la page de l’appel système *open*, située en section 2)
* **man –k open** (liste toutes les pages dont le titre contient *open*)

## Organiser ses sources : fichiers .c / .h. Exemple :



**Figure 1. Séparation des sources C en plusieurs fichiers**

**Pour comprendre les étapes de compilation/exécution d’un programme C et les types de fichiers impliqués :**



**Figure 2. Étapes de la compilation d’un programme C**

Démo : gcc -v / -E / -S / -c, discussion des résultats.

## 

## Exercices

*L’objectif de cette partie est de vous familiariser avec les fonctions de lancement d’exécutables (****exec****) et de traitement des erreurs (****perror****).*

***A savoir :*** Lors de chaque appel système, une erreur peut avoir lieu. L’erreur peut être détectée à l’aide de la variable **errno** (voir **man** **3 errno**) et affichée/expliquée à l’aide de la fonction **perror**. Tout programme écrit en TP doit prévoir le traitement des erreurs qui peuvent survenir lors de tous les appels système.

***Exemple*** *:* // appel système

f = open(...);

// traitement d’erreurs

if(errno) { // ou if(f < 0) cf. fonctionnement de open

perror("erreur open”); // affiche message d’erreur

\_exit(CODE\_SORTIE); // où CODE\_SORTIE != 0

}

1. **Lancement d’un exécutable à partir d’un programme C et traitement des erreurs**

Soit le programme suivant :

#include <stdio.h>

#include <unistd.h>

int main(int argc, char\* argv[]){

usleep(7000000); /\* sleep 7 seconds \*/

execl("/usr/bin/xterm", "xterm", NULL);

/\* si l'appel à exec fonctionne, le reste

du programme ne s’exécute jamais \*/

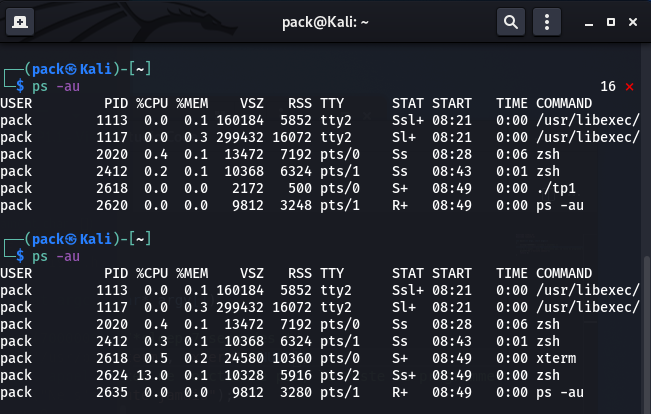
printf("ne s'affiche jamais");

return 0;

}

1. Compiler et lancer le programme ci-dessus et examiner son comportement. Dans un autre terminal, examiner la liste des processus (commande "**ps au**" pour obtenir une liste pertinente) et identifier ce processus avant/après l’exécution de la fonction **exec**. Que constatez-vous ?

Avant l’exécution de execl, nous pouvons voir un processus ./tp1 (nom de mon exécutable). Il possède le PID 2618. Après l’appelle à execl, le processus avec le PID 2618 à changer de nom et est devnue xterm.

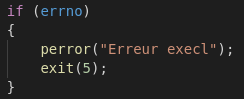


1. Modifier l’appel à **exec** pour passer en paramètre le chemin vers un exécutable inexistant. Que se passe-t-il ?

Si on met un exécutable inexistant, la console affiche « ne s’affiche jamais » et le processus se termine.

1. Traitement des erreurs : Dans le programme précédent, insérez un test d’erreur après l’appel à **exec**. En cas d’erreur, afficher le message d’erreur avec **perror** et terminez le programme avec l’appel à la fonction \_**exit** en utilisant un code d’erreur différent de zéro.

Pour la gestion d’erreur, nous rajoutons les lignes suivantes :



Lors de l’exécution, nous obtenons la ligne :  
« Erreur execl : no such file or directory ».

1. **Cycle de vie des processus**

**On reprend ici l’exercice 1 du TD. Pour rappel** : *Ecrire un programme C qui lance un processus fils (dont le code sera donné par le même programme), et qui produit les affichages suivants (les nombres seront les vrais PID lors de l'exécution) :*

* *affichages du processus père :*
  + *Je suis le père de 9762 : mon PID est 9761*
  + *Fin du père 9761*
* *affichages du processus fils :*
  + *Je suis le fils de 9761 : mon PID est 9762*
  + *Fin du fils 9762*

1. Modifier le code exécuté par le père afin qu'il se mette en sommeil pendant 30s *après* avoir lancé son fils, en utilisant la fonction **usleep()**. Visualiser l'état de ces processus avec **ps** ou **top**. Comment appelle-t-on ce type de processus fils ? A quelle condition pourront-ils disparaître de la liste des processus et du système ?

Ces processus disparaitront de la liste lorsqu’ils se réveilleront et se finiront.

1. A l'inverse : revenir à la version initiale du père, et modifier le code exécuté par le fils afin qu'il se mette en sommeil pendant 30s après le premier affichage. Visualiser ces processus avec la commande **"ps -ef"** (qui affiche aussi le PPID !). Dans la liste des processus de la machine, qui est alors le père du processus fils, une fois son père terminé ?

Lorsque le père s’est terminé, le fils à été adopté par 1081.

1. A la fin du processus fils, est-ce qu’il reste dans la liste affichée par **ps** ? Que pouvez-vous en déduire par rapport au comportement du "père adoptif" ?

Le processus fils ne reste pas affiché dans la liste. Le père adoptif continue donc le fonctionnement du programme.

1. **Contrôle d’un processus par envoi de signaux**

Pour cet exercice, utiliser le programme suivant :

#include <stdio.h>

#include <unistd.h>

int main(){

printf("Mon PID: %d\n", getpid());

while (1){

printf(".");

fflush(stdout);

usleep(1000000) ;

}

return 0; /\* jamais atteint \*/

}

1. Compiler puis lancer le programme dans un terminal.
2. Depuis un autre terminal, à l'aide de la commande **kill**, envoyer au processus le signal SIGSTOP, contrôler son état (**ps**), puis lui envoyer le signal SIGCONT. Que constatez-vous ?

Lorsque l’on envoie SIGSTOP au processus, il se met en pause mais est toujours affiché dans la liste des processus. Après l’envoie de SIGCONT, le processus reprend.

1. Tester ce qui se passe lors de l’envoi d’autres signaux au processus (voir le tableau suivant -- tester au minimum 3 signaux différents).

**Attention** : ne jamais utiliser les numéros de signaux en dur car ils peuvent changer d’un système à un autre (la liste ci-contre correspond à Mac OS X, les n° sont différents sous Linux). En échange, utiliser les noms symboliques des signaux, via une commande comme :

**kill -STOP 4567**

**No** **Name** **Default** **Action** **Description**

1 SIGHUP terminate process terminal line hangup

2 SIGINT terminate process interrupt program

4 SIGILL create core image illegal instruction

5 SIGTRAP create core image trace trap

9 SIGKILL terminate process kill program

11 SIGSEGV create core image segmentation violation

12 SIGSYS create core image non-existent system call invoked

13 SIGPIPE terminate process write on a pipe with no reader

14 SIGALRM terminate process real-time timer expired

15 SIGTERM terminate process software termination signal

17 SIGSTOP stop process stop (cannot be caught or

ignored)

18 SIGTSTP stop process stop signal generated from

keyboard

19 SIGCONT discard signal continue after stop

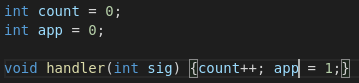
20 SIGCHLD discard signal child status has changed

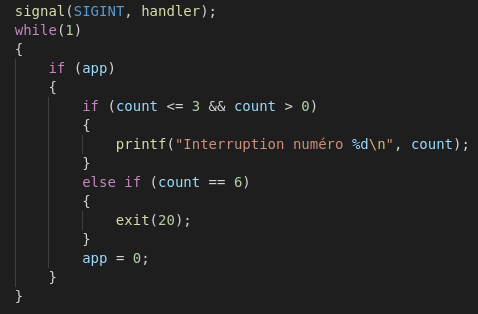
1. **Réception et gestion des signaux par les processus**

En faisant appel à la fonction **signal** (voir man) écrire un programme C dont le comportement à la réception du signal SIGINT (CTRL+C) soit le suivant :

1. A la première réception de 'CTRL+C' : on affichera "interruption numéro 1".
2. A la seconde réception de 'CTRL+C' : on affichera "interruption numéro 2".
3. A la troisième réception de 'CTRL+C' : on affichera "interruption numéro 3".
4. A la quatrième : rien.
5. A la cinquième : rien.
6. Enfin à la sixième : on terminera le processus.

Voici le code :





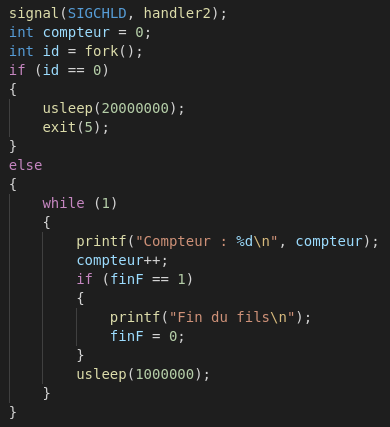
1. **Gestion de fils par SIGCHLD**

Ecrire un programme qui lance un processus fils puis vaque à ses occupations à l’infini (par exemple, il affiche en boucle un compteur, entre deux **sleep**). Le processus fils se met en sommeil pendant un certain temps (20 sec) puis se termine avec un code quelconque.

Nous souhaitons que le père puisse faire son travail en parallèle du fils, mais qu’il affiche toutefois un message lorsque le fils se termine. Il ne peut donc pas utiliser la fonction **wait** dans le programme principal car cela le bloquerait.

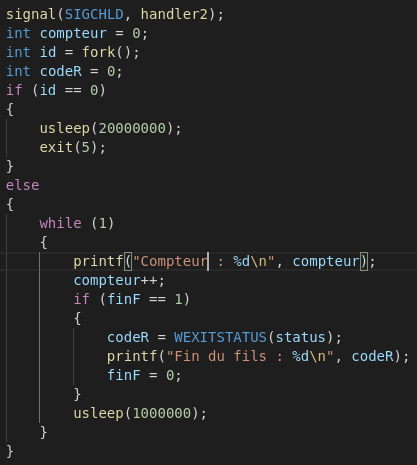
1. Traiter le signal SIGCHLD pour afficher le message quand le fils se termine





1. Dans le message, inclure le code de retour du fils (récupéré avec **wait** et **WEXITSTATUS**)





1. Que se passe-t-il si le fils est terminé par un signal (par exemple SIGTERM) ? Comparez ce comportement avec le cas où le fils est seulement suspendu (par SIGSTOP depuis un autre terminal) *avant sa fin.* Pourquoi dans ce cas le processus parent semble-t-il suspendu également ?

Lorsque l’on met en pause le fils par un SIGSTOP, le père arrête son fonctionnement et se bloque. Si l’on arrête le fils par un SIGTERM, le programme affiche le message « fin du fils : 0 » et continue son fonctionnement.

1. Remplacer l’appel à **wait** par :  
    waitpid(-1, &status, WNOHANG);  
   Cela marche-t-il mieux en cas de SIGSTOP sur le fils ? Expliquer pourquoi.

Lors de l’exécution de SIGSTOP, le programme affiche « Fin du fils : 0 » . Le WNOHANG permet d’arrêter le blocage du processus