R3.05 - TP 3
Création de Processus

La fonction system()

Premières expérimentations

La fonction **`system()`** est la manière la plus simple de créer un processus depuis un autre processus. “La plus simple” ne signifie pas qu’elle est idéale et que c’est celle que vous utiliserez tout le temps car elle a quelques inconvénients, donc le principal est de mettre en pause le processus initiateur de l’appel. On perd alors le côté multitâche.

Question 1

Tapez le code suivant dans votre éditeur favori :

```
#include <stdlib.h>
int main() {
     system("ls");
     return EXIT_SUCCESS;
}
```

Compilez-le et exécutez-le.

Vous l’aurez deviné (j’espère), ce code affiche la liste des fichiers du répertoire courant.

Question 2

Modifiez votre code pour exécuter un **`ls -l`** maintenant. Compilez et exécutez. Cela fonctionne-t-il ? Si vous n’y parvenez pas, appelez votre enseignant.

Maintenant modifiez encore votre code pour envoyer le résultat du **`ls -l`** dans un fichier de votre choix. Suivez votre intuition, ça doit être la bonne, sinon faites encore appel à votre enseignant.

A ce stade, vous devez avoir réussi à créer (par appel **`system()`**) un fichier contenant la liste des fichiers du répertoire courant. En principe, vous devriez avoir utilisé une redirection pour y arriver. Cependant, ce mécanisme de redirection est un mécanisme qui est propre au Shell. Il paraît donc étrange qu’un appel système (donc indépendant d’un

Shell) aboutisse à un tel résultat. Enfin, vous n'êtes peut être pas étonné, mais vous devriez l'être. Expérimentons un peu plus pour en comprendre la raison.

Comment cela fonctionne-t-il ?

Question 3

Sur la base du code précédent, affichez la liste de vos processus (**`ps -wafu`**) depuis votre programme.

Compilez, exécutez. Qu'observez-vous ? Quels sont le(s) processus créé(s) par votre exécution ? Comprenez-vous maintenant pourquoi la redirection qui, rappelons-le, est un mécanisme du Shell, fonctionne dans votre programme qui utilise seulement un appel **`system()`** ? Encore une fois, faites-vous expliquer ça par votre enseignant si vous ne comprenez pas ce qui s'est passé.

Une séquence de commandes

Question 4

Partant des constatations et expérimentations précédentes, proposez un programme effectuant, **en un seul** appel **`system()`**, l'affichage du nombre d'objets (fichiers ou dossiers) dans le répertoire **`HOME`** de l'utilisateur qui lance la commande (ça fait appel à votre mémoire de SYS 1A). Lisez bien l'intitulé de ce paragraphe, ça peut vous aider !

Pas de multitâche

Pour expérimenter comment se comporte votre programme quand il appelle **`system()`**, reprenez la base de code qui nous sert depuis le début de ce TP et apportez les modifications suivantes :

- Placez une ligne qui affiche (avec un **`printf()`**) le texte **`"Avant"`** avant l'appel à **`system()`**
- Placez une ligne qui affiche **`"Apres"`** après l'appel à **`system()`**
- Exécutez une commande **`sleep 5`** à l'aide d'un appel à **`system()`**

Compilez, exécutez, observez l'affichage. Comment se comporte votre programme quand il est dans l'appel **`system()`** ?

Conclusion

La fonction **`system()`** est un moyen rapide et simple de créer un processus mais a un inconvénient majeur : elle est bloquante !

La fonction **`system()`** ne crée pas réellement elle-même un nouveau processus mais fait appel à un mécanisme de création que nous allons maintenant découvrir ensemble et qui est un fondement essentiel d'Unix. Tous les processus créés le sont de cette façon là.

Création d'un processus Unix

Genèse

Comme dans la nature vivante, rien ne naît du néant. Tout est engendré par un géniteur ou une génitrice : une plante, un animal ou un être humain, par exemple.

Bon, techniquement, pour le monde animal (incluant l'homme), il y a souvent deux géniteurs, mais on va simplifier le procédé en considérant l'embryon comme le début du clonage. On est en BUT Informatique 👩‍💻 pas en médecine 🧑‍⚕️!

Sous Unix, c'est la même chose. Tout processus Unix naît d'autre chose. Cette autre chose est simplement un processus qui engendre une copie de lui-même. Cette copie devient un nouveau processus qui est l'exacte copie de son processus géniteur. C'est donc une sorte de clonage, comme on pourrait cloner un être vivant, quelque chose de techniquement possible, même si en France le clonage est interdit sur l'humain.

Revenons à Unix, car c'est ce qui nous importe aujourd'hui !

A partir du moment où un processus a créé une copie de lui-même, cette copie devient alors autonome, et va vivre sa vie de processus de son côté, indépendamment de son processus géniteur. Le processus géniteur va aussi vivre sa vie de son côté, en laissant son clone vivre la sienne. Comme dans la vraie vie entre parents et enfants finalement ! On reviendra plus loin sur le fait qu'il existe quand même un lien entre ces deux-là (sous Unix et dans la vraie vie).

Le processus issu du clonage apparaît alors dans la liste des processus avec son propre **`PID`**, comme un enfant vit sa propre vie et possède son propre numéro de sécurité sociale (son **`PID`** d'humain).

On ne connaît pas encore les détails du clonage de processus mais à ce stade de votre lecture, si ce qui est écrit n'est pas clair pour vous, demandez à votre enseignant.e un peu de ses lumières.

La fonction fork()

Analogie routière

Tout se joue avec une fonction système qui assure le clonage du processus courant.

La fonction s'appelle **`fork()`** qui signifie, en anglais, bifurquer, faire un embranchement. On voit bien dans cette analogie routière qu'on va se retrouver, après l'appel à **`fork()`**, avec deux routes séparées alors qu'on venait du même endroit, de la même route.

Après **`fork()`** les deux "routes" sont simplement deux processus, le processus géniteur et son clone.

On peut dire, avec l'analogie routière, que la route d'où on vient se poursuit après l'embranchement et que la seconde route est finalement une nouvelle route ajoutée à

l'itinéraire. De la même façon, après un **`fork()`** il reste le processus géniteur qui continue "sa route" et le clone va suivre la "nouvelle route ajoutée".

Dans un premier temps, gardons cette analogie routière et posons-nous les questions suivantes :

- À partir de quel endroit débute la seconde route ?
- Si un véhicule emprunte la seconde route, peut-on considérer que la route avant l'embranchement fait aussi partie du chemin global qu'il aura fait ?

La réponse à la première question est évidemment que la seconde route débute après l'embranchement. Un véhicule qui souhaite utiliser cet embranchement ne va pas repartir de son point de départ quand il va arriver à cet embranchement, ça va de soi !

La réponse à la seconde question est assez évidente mais mérite qu'on s'y concentre bien, car de la bonne compréhension de cette question et de sa réponse va découler la bonne compréhension de ce qui se passe avec un processus Unix.

Ainsi, si un véhicule emprunte l'embranchement de gauche, le conducteur considérera que le chemin avant l'embranchement + le chemin après l'embranchement de gauche sont globalement un seul et même chemin, pour lui, pour son voyage. Dans sa tête il se rappellera de son voyage global sans encombrer son esprit des autres chemins qu'il n'aura pas suivi.

Si on se place maintenant dans le cas d'un conducteur qui fait l'autre choix de prendre le chemin de droite, son expérience et son souvenir de voyage seront similaires à l'autre conducteur, **il aura des souvenirs communs pour la portion commune du chemin (avant l'embranchement) mais des souvenirs différents et personnels sur la partie après l'embranchement**.

Sous Unix

Sous Unix, avec les processus, on retrouve exactement cette analogie routière :

- Un clone va commencer sa vie, non pas en retournant et en redémarrant au début de la route globale (le **`main()`**), mais directement après l'embranchement, c'est-à-dire après le moment où il a été cloné par le processus géniteur. On peut ainsi considérer que sa vie globale est composée d'un début de vie qui est celle de son géniteur + sa propre vie après le clonage. C'est peut-être contre-intuitif d'un point de vue biologique mais nous sommes des clones de nos parents et nos gènes traduisent aussi nos héritages issues des vies de nos ancêtres (Freud, sors de ce corps !).
- Un clone va donc hériter de son expérience, de la mémoire du voyage avant l'embranchement, c'est-à-dire avant son clonage.
- Un clone va ensuite vivre sa vie sur son chemin et le processus géniteur en fera de même sur le sien, chacun ayant, à partir de ce moment (après le clonage), sa propre expérience de son voyage, de son exécution.
- Clone et géniteur ont une expérience commune du voyage avant le clonage et une expérience individuelle d'après le clonage.

Pratiquons

D'un accès moins immédiat que la fonction **`system()`**, l'usage de **`fork()`** est totalement différent. C'est son côté multitâche qui est intéressant.

Terminologie

Avant de commencer, mettons au clair la terminologie que nous allons utiliser à partir de maintenant :

- Le processus géniteur s'appelle le **processus parent** ou **processus père**.
- Le processus clone s'appelle le **processus enfant** ou **processus fils**.

Code de base

Voici un premier code de base d'un programme utilisant **`fork()`**.

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int main() {
     pid_t    pid;
     int      val = 10;

     printf("Avant fork(), je suis PID %d", getpid());           ①
     printf(" et val = %d\n", val);      ②
     pid = fork();                       ③
     if (pid == 0) { // Fils             ④
          //sleep(1);
          printf("Je suis le fils, mon PID est %d et mon pere a le PID
%d\n", getpid(), getppid());
          printf("Chez le fils, val = %d\n", val);
          val = 20;                      ⑤
     } else { // Pere                    ⑥
          printf("Je suis le pere, mon PID est %d et je viens de creer
un fils de PID %d\n", getpid(), pid);
          printf("Chez le père, val = %d\n", val);
          sleep(1);
     }
     printf("Je suis PID %d et val = %d\n", getpid(), val);       ⑦
     return EXIT_SUCCESS;                ⑧
}
```

Compilez-le et exécutez-le. Vous observez que le père et le fils affichent leur “livret de famille” respectif et que les valeurs sont cohérentes entre père et fils.

Explication du code de base

① et ② : Tout ce qui est avant **`fork()`** fait partie de la vie du processus père. Seul ce processus père exécute cette partie car pour le moment il n’y a pas encore de fils, la “naissance” du processus fils n’a pas encore eu lieu. Le processus fils n’exécutera jamais cette partie car il commencera sa vie après l’instruction qui effectue son clonage.

③ : Le clonage à lieu ici, avec la fonction **`fork()`**. En retour de l’appel de la fonction **`fork()`**, on se retrouve avec 2 processus, le père et le fils.

Chacun des deux continue sa vie, sa route, en retour de l’appel de la fonction **`fork()`**, c’est-à-dire que chacun des deux va affecter sa propre variable **`pid`** avec ce que **`fork()`** a retourné. **`fork()`** va retourner à chacun une valeur différente qui est essentielle pour différencier le père de son fils car, comme tout clone, chacun va avoir l’impression d’être le processus originel !

À priori, rien ne permet de différencier quel est celui qui a cloné l’autre, chacun ayant le même souvenir du voyage qui l’a mené jusqu’à cet endroit. Il faut donc avoir une façon (dans le code C) de les différencier, que le code puisse être capable de traiter le fait d’être le père ou le fait d’être le fils :

- En retour de la fonction **`fork()`** le père reçoit le **`pid`** du fils créé.
- En retour de la fonction **`fork()`** le fils reçoit **`0`**.
- La fonction **`fork()`** retourne **`-1`** pour signifier une erreur de **`fork()`** et dans ce cas aucun processus enfant n’a été créé. errno contient, comme toujours, un code indiquant le problème. C’est un cas rare mais qui peut arriver et qui est souvent lié au fait d’avoir atteint une limite du nombre de processus qu’un utilisateur a le droit de lancer ou un problème de mémoire consommée.

  Dans un souci de simplification, ce cas (**`-1`**) n’est pas traité ici. Les limites sont assez larges, ce sont donc des cas extrêmes, mais il faudra en tenir compte quand même !

Ainsi, en ④ et ⑥, cette valeur de retour de la fonction **`fork()`** permet de différencier qui est le père de qui est le fils.

> Attention, même si la variable **`pid`** a été définie au départ dans le père, chaque processus va disposer de **ses propres variables** tout en héritant des valeurs présentes dans ces variables au moment du **`fork()`**.
>
> Toute modification de ces variables après le **`fork()`**, par le père ou par le fils, sera personnelle au processus qui effectue la modification, sans impacter les variables de l’autre processus. Ainsi la variable **`pid`** du père aura sa valeur (qui est retournée par **`fork()`**) et celle du fils une autre valeur (aussi retournée par **`fork()`**).

④ : Nous sommes dans le code qui concerne le “voyage” du fils.

⑤ : Le fils modifie sa propre variable **`val`**, sans impact sur celle de son père.

⑥ : Nous sommes dans le code qui concerne le “voyage” du père.

⑦ et ⑧ : Même si ce code n’est pas traité spécifiquement dans le bloc “père” ou le bloc “fils”, le fait qu’il soit placé après le **`fork()`** implique qu’il est donc commun au code des deux processus et sera exécuté par chacun de ces deux processus, mais avec chacun sa propre variable **`val`** et son propre **`pid`**.

Vous aurez noté que **`getpid()`** est une fonction système permettant à un processus de connaître son propre **`PID`**. Cependant, il s’agit bien de la valeur de retour de **`fork()`** qu’il faut tester pour différencier le père (retour **`> 0`**) du fils (retour **`== 0`**) car, même dans le fils, **`getpid()`** retournera une valeur de **`PID`** car tout processus possède bien un **`PID`**.

`getppid()` permet de connaître le PID du père.

Exercice 2

Sur la base du code proposé précédemment, et comme vous avez appris à traiter les erreurs des appels aux fonction système, ajoutez le traitement d’une erreur de **`fork()`**.

Ce sera difficile pour vous de tester votre code car il faudrait se trouver dans des conditions que vous ne rencontrez que très rarement. Vous pouvez éventuellement “forcer la (mal)chance” en codant un changement manuel de la valeur retournée par **`fork()`**. Attention, faites un traitement adapté en cas d’erreur, généralement on met un terme au processus avec un joli message d’erreur.

Un père doit s’occuper de ses enfants

Modifiez le code précédent :

- Enlever le commentaire du 1er **`sleep(1)`**
- Mettre en commentaire le 2ème **`sleep(1)`**

Compilez, exécutez, observez. Que se passe-t-il de différent cette fois-ci ? L’affichage des relations père-fils est-il toujours cohérent ? Regardez le **`PID`** du père sur la ligne **`Je suis le fils ...`** . Qu’observez-vous ? Si vous ne voyez pas la différence, faites appel à votre enseignant.e.

Règle d’Or

Un processus qui engendre un ou plusieurs autres processus par un ou des **`fork()`** doit impérativement attendre la mort de tous les processus qu’il a engendrés. C’est seulement quand il aura attendu qu’il pourra lui-même s’arrêter.

La fonction système qui attend la mort d’un processus enfant est **`wait()`** ou **`waitpid()`**. Vous consulterez le manuel pour plus de détails le moment venu.

`wait()` attend la mort de n’importe lequel de ses enfants alors que **`waitpid()`** attend celle d’un enfant dont on spécifie le **`PID`** en paramètre (voir le **`man`**).

Sauf exception, on effectue généralement plutôt des **`wait()`** car **`wait()`** et **`waitpid()`** sont, en principe[1], bloquants, dans l'attente de l'événement de cette mort. Donc si on a plusieurs processus enfants, comme on n'a aucune certitude concernant l'ordre dans lequel ces enfants vont s'arrêter, un **`waitpid()`** pourrait bloquer alors qu'un autre processus enfant est mort avant celui qu'on est en train d'attendre avec le **`waitpid()`**.

Cette action d'attente avec **`wait()`** ou **`waitpid()`** permet d'indiquer au système qu'on a pris connaissance de la mort d'un mort d'un processus enfant.

Si le père ne fait pas cette action, il se passe quelque chose qui va être expliqué dans l'exercice **The Walking Dead** un peu plus loin.

Par contre, si le processus parent s'arrête avant ses enfants, les processus enfants changent de père et deviennent des descendants direct d'un processus particulier qui s'appelle **`init`** ou **`systemd`**[2] et qui a en principe le **`PID 1`** ! Ce que vous devez observer dans l'exercice précédent. Vérifiez.

Note **`wait()`** attend un paramètre dont on n'a pas forcément besoin. Vous pouvez passer **`NULL`** dans ce cas. C'est notamment le cas dans cet exercice.

Corrigez maintenant ce qui manque dans le père et uniquement lui. Ne touchez rien aux lignes **`sleep(1)`** et ajoutez la fonction **`wait()`** manquante au père. Afin de savoir où on est et ce qui se passe, placez aussi des **`printf()`** avant et après l'appel de ce **`wait()`**. Testez que ça fonctionne de nouveau correctement. Vérifiez notamment le message **`Je suis le fils ...`** . Relisez ce qui précède dans ce chapitre et/ou faites appel à l'enseignant.e pour vous aider si besoin.

Code de base final

Quand vous aurez fait fonctionner le code précédent, vous pourrez retirer les **`sleep(1)`** (supprimez les lignes ou mettez-les en commentaire). Ce code vous servira de base pour tout programme utilisant **`fork()`**.

Gardez-le précieusement.

Lors des contrôles, vous ne serez pas jugé sur la similitude de cette partie précise de vos codes (profitez-en, c'est pas tous les jours cadeau), puisqu'on vous a guidé à la construire ainsi, mais si ce n'est pas codé correctement, vous n'aurez plus aucune excuse et, par contre, cela sera jugé ! 😉

Les fonctions exit() et _exit()

Il existe deux fonctions permettant à un processus de se terminer immédiatement.

Non, on ne fait pas un **`kill(SIGKILL, getpid())`** !!!

Il faut utiliser **`exit()`** ou **`_exit()`**. Elles sont presque identiques mais **`exit()`** ne s'utilise que dans le cadre d'un processus parent et **`_exit()`** dans les processus enfants lorsque

[1] Il y a une option qui permet de ne pas bloquer, on la verra plus tard.
[2] Le TP 3 sur les processus en BUT1 doit vous rappeler quelque chose (**`ps -axuwf`**, **`pstree`** ?)

les enfants héritent de fichiers ouverts par le parent. En effet, **`exit()`** ferme tous les fichiers ouverts et que vous n'avez pas fermés vous-même (ce qui n'est pas bien, hein ?!). De ce fait, si un enfant utilise **`exit()`** il va aussi fermer les fichiers ouverts par son parent et dont il a hérité lors du **`fork()`**, ce qui n'est pas souhaitable. En utilisant **`_exit()`**, cette partie du nettoyage n'est pas faite par l'enfant, et c'est ce qu'on souhaite en général puisque le parent s'occupera en principe de cette tâche.

Le signal SIGCHLD

The Walking Dead

Reprenez votre code de base final de l'exercice précédent.

IMPORTANT : si vous avez un **`wait()`** dans ce code de base final[3], mettez-le en commentaire pour l'expérimentation ci-dessous.

Modifiez votre code pour que le père fasse, juste immédiatement après l'affichage de **`Je suis le père`**, une boucle de **`1`** à **`300`** et qu'à chaque passage il fasse un affichage **`Passage N`** (où **`N`** est la valeur de la variable de boucle) suivi d'une pause (**`sleep`**) de **`1`** seconde.

Le fils ne fait rien de plus que ce qui est déjà dans le code de base.

Compilez, exécutez. Dans un second Terminal, lancez un **`ps <PID>`** avec le **`PID`** du fils. Qu'observez-vous concernant le fils dans la colonne **`STAT`** ?

Explication : tant que l'action **`wait()`** n'a pas été faite par le processus père, le processus enfant reste inactif dans la liste des processus sous la forme d'un processus **Zombie**. Vous avez dû observer qu'il y a un **`Z`** dans la colonne **`STAT`** de la commande **`ps <PID>`**.

Faire un **`wait()`** est la solution, sauf que le père a peut-être autre chose à faire que d'attendre qu'un processus enfant s'arrête (meure) ! Le principe de **`fork()`**er n'est pas de faire travailler les enfants et d'attendre sagement la fin. L'idée est souvent de distribuer du travail et de travailler soi-même, ce que va certainement faire le processus père aussi.

Mais dans ce cas, comment le processus père peut-il savoir qu'un processus enfant est mort et qu'on attend de ce père qu'il fasse un **`wait()`** de son enfant, pour retourner continuer de travailler ensuite, voire même qu'il relance un nouveau processus enfant pour remplacer le mort ?

Il serait pratique, pour le père, d'être averti juste au bon moment, pour faire un **`wait()`** opportun sans perdre de temps inutile.

Vous avez noté qu'on vient d'utiliser l'expression "être averti" ? Ça n'éveille pas une idée avec quelque chose qu'on a expérimenté dans le TP précédent ?

[3] Et vous devriez, en principe, en avoir un…

Utiliser un signal

Pour résoudre le problème soulevé dans l'exercice précédent, vous allez faire appel à ce que vous avez vu dans le TP sur les signaux.

Cherchez dans le **`man 7 signal`**, un signal qui pourrait être notre solution.

Mettez en place un détournement de ce signal et dans la fonction de traitement du signal, faites l'appel nécessaire pour résoudre le problème de l'exercice précédent.

Encore une fois, faites appel à l'enseignant.e pour confirmer votre idée ou pour débloquer la situation si vous avez un doute ou besoin d'un coup de main.

Et si vous n'y arrivez vraiment pas, fuyez avant que tous les processus Zombies créés autour de vous ne vous rattrapent !