TP 08 : Fork & co

Introduction: Dans notre première séance de programmation système, on va principalement s'intéresser à la commande `fork`.

1 Prélude

Pour écrire un programme C utilsant `fork`, il faut inclure les librairies suivantes dans vos programmes :

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

Vous aurez probablement par la suite besoin aussi des librairies suivantes :

```
#include <sys/types.h>
#include <sys/wait.h>
```

2 Forkons entre amis

Un programme simple Le programme suivant montre une utilisation basique de `fork`.

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

int main() {
  printf("Bonjour,\n");
  fork();
  printf("J'ai fait un 'fork' !\n");
  return 0;
}
```

- À votre avis, qu'affiche ce programme ? Tester pour vérifier le comportement attendu.
- Écrire un programme qui étant donné une MACRO `BOUND`, créer 2^{BOUND} processus
- Que fait la commande `pstree` ? Pouvez-vous observer le comportement de votre programme à l'aide de cette commande ? Pourquoi ?
- Modifier très légèrement votre programme afin d'observer le comportement de votre programme à travers `pstree`.
- Il est possible de tuer un processus à l'aide de la commande `kill`, comment ?
- Observer l'effet de la commande `kill` sur votre programme à travera `pstree` ? Pourquoi à votre avis certains processus sont *vraiment* tués et pas d'autres ?
- En utilisant la fonction `wait`, corriger le problème identifié à la question précédente.
- Modifier votre programme pour afficher les PIDS des processus crées. L'ordre est-il toujours le même ? Pourquoi ?
- (Bonus) Expliquer le comportement de la commande suivante sans l'éxécuter (il est même fortement déconseillé de l'éxécuter) :

```
:(){ :|:& };:
```

— (Bonus) Faire un programme paramétré par un entier n qui lance exactement $n!$ processus

Tips and tricks

— Quelle est la sortie attendue du programme suivant ? Éxécuter puis vérifier votre conjecture.

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

int main() {
    printf("Je vais faire un 'fork'... ");
    int p = fork();
    if (p == 0) {
      printf("Je suis le fils...\n");
    } else {
      printf("Je suis le père...\n");
    }
    return 0;
}
```

— Que se passe t'il lorsqu'un processus fait un fork(), et que le père et le fils lisent sur l'entrée standard (avec getchar() par exemple) ? Décrivez votre méthodologie ainsi que les résultats trouvés.

— Écrivez un programme qui se comporte de la manière suivante :
  — le processus initial créé un fils et se termine,
  — le fils créé un fils (petit fils du processus initial) et se termine,
  — le petit fils créé un fils (petit petit fils du processus initial) et se termine,
  — etc...

  Expliquez pourquoi le programme obtenu ne sature pas la table des processus.

— Est-ce que le processus enfant peut tuer le processus père ? Comment ?

Attendre patiemment Expliquer les différences de comportement entre les trois programmes suivants (je vous invite à utiliser la commande `top` pour observer les différences) :

```c
int main() {
while(1);
return 0;
}
```

```c
#include <stdio.h>
int main() {
  getchar();
  return 0;
}
```

```c
#include <unistd.h>
int main() {
  while(1) {
    sleep(1);
  }
  return 0;
}
```

Diviser pour mieux attendre Tester le programme suivant :

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    int p = getpid();
    printf("DÉBUT %i\n", p);
    unsigned n = 0x7fffffff;
    while(n--);
    printf("FIN %i\n", p);
    return 0;
}
```

puis ajuster la valeur de `n` pour que l'éxécution prenne environ entre 5 et 10 secondes.

— Que se passe-t-il si vous lancez deux fois ce programme en parallèles ? Quelle est la durée d'exécution totale de ces deux processus ? Refaire l'expérience pour 4 processus et prédire le résultat.

— Que se passe t'il si on remplace la boucle `while(n--);` par un `sleep(5);` ?

3 Calculons

Dans cet exercice on étudie l'utilisation et l'évaluation du code de sortie d'un processus. Normalement, le code de sortie est utilisé pour indiquer par exemple, la résussite ou non d'un programme. Ici, on s'en sert pour communiquer les résultats d'un calcul. (Remarque : le code de sortie doit entre entre 0 et 255, donc ce n'est pas idéal pour ce genre de chose (c'est juste un exemple !). Pages man utiles : `fork(2)`, `wait(2)`. Utilisez `make` pour compiler.

— On commence avec un exemple simple `simple.c` qui calcule la somme de deux entiers. Complétez le programme tel que le fils utilise `exit` pour communiquer la somme au père et que le père reçoit cette valeur par `wait`.

— Application plus complexe : Télécharger l'archive `calc.zip` qui contient un calculateur simple qui évalue des expressions avec des entiers naturels, addition, multiplication et subtraction. Dans l'état actuel le programme convertit l'expression vers un arbre, puis il traverse les noeuds de l'arbre un par un pour calculer les valeurs de toutes les sous-expressions. Votre tâche est de permettre au programme de calculer les sous-expressions en parallèle. Lorsque le programme évalue une sous-expression, il devrait lancer deux processus fils qui évaluent leurs sous-expressions et qui on communiquent le résultat au père par leurs codes de sortie. Le père attend les deux résultats et applique l'opération correspondante pour obtenir son propre résultat. Il suffit de modifier la fonction compute dans main.

4 Ping-Pong

Écriver un programme qui créer un processus fils et qui à le comportement suivant : le père envoie un par un, 1 bit d'information au processus fils. Ce dernier doit afficher la suite de bits émise par son père **dans l'ordre**. Pour se faire, il vous est demandé d'utiliser les signaux. La difficulté de cet exercice est que l'ordre n'est pas garanti : un signal A émit avant le signal B par le processus père peut être reçu dans l'ordre inverse par le processus fils.