L'interface système UNIX

1 Manipulation des appels système

Exercice 1 *Que fait le code suivant et pourquoi ?*

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

int main(int argc, char const *argv[])
{
    printf("Dauphine");
    write(1, "MIDO", 4);
    printf("paris\n");
    return 0;
}
```

Que se passe t'il si on ajoute un retour à la ligne après Dauphine et pourquoi ?

Exercice 2 *Écrire un programme qui ouvre un fichier en écriture avec* `fopen` *et écrit 1e7 fois le caractère '0' dans ce fichier avec la fonction* `fwrite`.

Écrire un autre programme qui ouvre un fichier en écriture avec `open` *(appel système) et écrit 1e7 fois le caractère '0' dans ce fichier avec l'appel système* `write`. *(Utiliser la page de manuel* `man 2 open` *et* `man 2 write`*)*

Comparer les temps de calcul avec la commande time

Exercice 3 (`exo_1.c : open, read, write, close`**)** *Écrire un programme qui recopie un fichier. Cette commande a deux arguments et recopie le premier fichier (dont le nom est le premier argument) dans le second (dont le nom est le second argument).*

Exercice 4 (`exo_2.c : open, read, write, close, dup`**)** *Écrire un programme qui recopie un fichier sur la sortie standard ou dans un autre fichier. S'il n'y a qu'un seul argument (c'est le nom du fichier à recopier), on envoie le fichier sur la sortie standard ; s'il y en a deux, on redirige la sortie standard dans un fichier dont le nom est le second argument :*

```
si argc < 2
      erreur
ouvrir fichier d'entrée
si argc == 3
      ouvrir fichier sortie
      rediriger le descripteur 1 vers le fichier
boucle de copie avec écriture vers descripteur 1
```

Exercice 5 (`exo_3.c fork, getpid, getppid`**)** *Pour comprendre le fonctionnement de l'appel système* `fork`*, rédiger un programme implémente le pseudo code suivant:*

```
si (fork() == 0)
    afficher "Le bloc 'si' est exécuté."
sinon
    afficher "Le bloc 'sinon' est exécuté."
```

Compiler et exécuter ce programme. Qu'observe-t-on? Ce résultat est-il surprenant? Lire attentivement le `man 2 fork` *et tenter de proposer une explication. Répéter l'opération en utilisant* `getpid` *et* `getppid` *pour savoir quel processus et responsable de quel affichage, puis affiner votre explication.*

Exercice 6 (`exo_4.c : fork, getpid, getppid, wait`**)** *Écrire un programme qui se duplique. Ensuite, le père affiche son numéro et celui de son fils, le fils affiche le sien et celui de son père. Pour finir, le père attend le fils et récupère, en l'affichant, la valeur de retour envoyée par le fils lors de sa terminaison.*

Exercice 7 (`exo_5.c : execvp`**)** *Écrire un programme qui utilise l'appel système* `execvp` *pour lancer l'exécution de* `wc -l exo_5.c`*.*

Exercice 8 (`exo_6.c : fork, execvp, wait`**)** *Dupliquer le processus avec* `fork`*. Après la duplication le fils exécute la commande composée du reste des arguments de la ligne de commande. (Exemple : pour* `exo_6 wc -l`*, le fils exécute la command* `wc -l`*.)*

Exercice 9 (`exo_7.c : fork, open, read, write, close, pipe`**)** *Combiner* `exo_2` *et* `exo_6` *: le père ouvre un fichier (dont le nom est en paramètre) et l'envoie dans un tube, le fils lit le tube et recopie les données reçues dans un autre fichier (dont le nom est également passé en paramètre). (Attention à la fermeture du tube sur les extrémités non utilisées.)*

Exercice 10 (`exo_8.c : fork, execlp, pipe, dup, read/fputs, write/fgets, close`**)** *Écrire une commande ayant un argument qui est sous la forme d'une chaîne de caractères (par exemple* `exo_7 ls | wc -l`*). Le processus crée un tube, se duplique et redirige la sortie standard du fils dans le tube. Le processus fils exécute la commande et pendant ce temps, le processus père récupère l'extrémité du tube et lit ce que le fils y écrit avant de l'afficher à l'écran.*