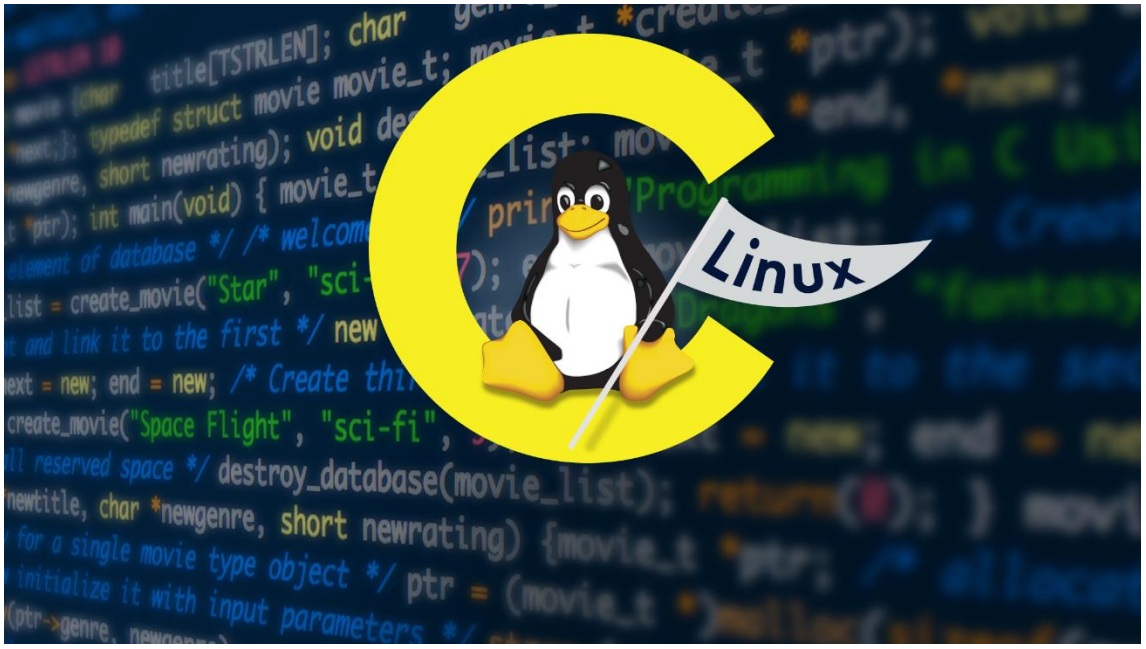
5/18/2019

Projet Os (Pratique)

Linux – Langage C



NIYONKURU Olivier - VALENTIN Morgan - AUTOME Edwin
EPHEC

Table des matières :

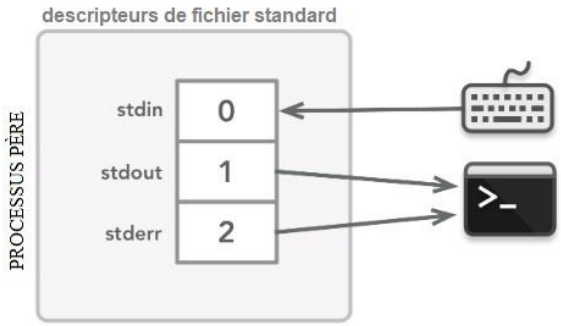
Introduction :

Lors du cours système d'exploitation, il nous est demandé de réaliser un code dans le langage de programmation C sous le système d'exploitation linux. Ce code reçoit via la ligne de commande une commande avec ses options / paramètres de sorte qu'en exécutant le code (compilé) suivie d'une commande, celui-ci créer un deuxième processus dit fils qui va afficher le résultat de l'exécution du processus père.

Analyse du travail :

1. Lancement du programme

Lorsque le programme est lancé, le processus parent est créé avec trois descripteurs de fichiers ouverts par défaut. *STDIN* est attaché au clavier et *STDOUT* et *STDERR* apparaissent dans le terminal. Cependant, nous pouvons rediriger *STDOUT* ou *STDERR* à tout ce dont nous avons besoin.



Les flèches indiquent le flux de données : **stdin** reçoit une entrée du clavier et **stdout** et **stderr** envoient une sortie à l'affichage du terminal. Ils sont identifiés comme ça :

- STDIN FILENO ou 0 : l'entrée standard
- STDOUT FILENO ou 1 : la sortie standard
- STDERR FILENO ou 2 : la sortie d'erreur

2. Création du pipe

Un tube de communication est un tuyau (en anglais ***pipe***) dans lequel un processus peut écrire des données et un autre processus peut lire. Dans notre cas, le fils va écrire dans le tube et le parent va lire dans le tube. On crée un tube par un appel à la fonction pipe, déclarée dans « ***unistd.h*** ».

L'instruction ***pipe()*** prend un argument qui est un tableau de deux entiers. Elle met dans le premier entier la valeur du descripteur de fichiers pour la lecture, et dans le second la valeur du descripteur de fichiers pour l'écriture. Ainsi, tout ce qui est écrit via le premier descripteur de fichiers est disponible en lecture via le second descripteur de fichiers.

La fonction renvoie -1 en cas d'erreur et 0 si elle réussit, et elle crée alors un nouveau tube. La fonction pipe remplit le tableau descripteur passé en paramètre, avec :

- descripteur[0] désigne la sortie du tube (dans laquelle on peut lire des données) ;
- descripteur[1] désigne l'entrée du tube (dans laquelle on peut écrire des données) ;

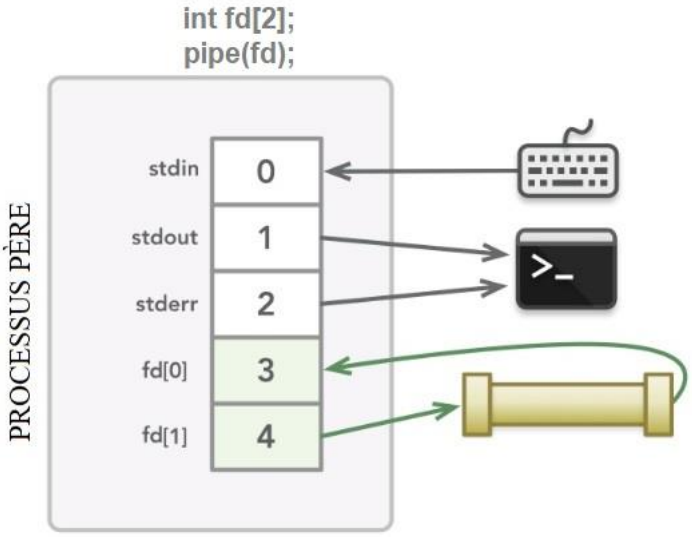
Le principe est qu'un processus va écrire dans descripteur[1] et qu'un autre processus va lire les mêmes données dans descripteur[0].

Le problème est qu'on ne crée le tube dans un seul processus, et un autre processus ne peut pas deviner les valeurs du tableau descripteur.

Pour faire communiquer plusieurs processus entre eux, on fait appeler la fonction pipe avant d'appeler la fonction ***fork()*.**

Ensuite, le processus père et le processus fils auront les mêmes descripteurs de tubes, et pourront donc communiquer entre eux.

De plus, un tube ne permet de communiquer que dans un seul sens. Si l'on souhaite que les processus communiquent dans les deux sens, il faut créer deux pipes, mais pour le travail demandé on a choisi de créer un seul tube.



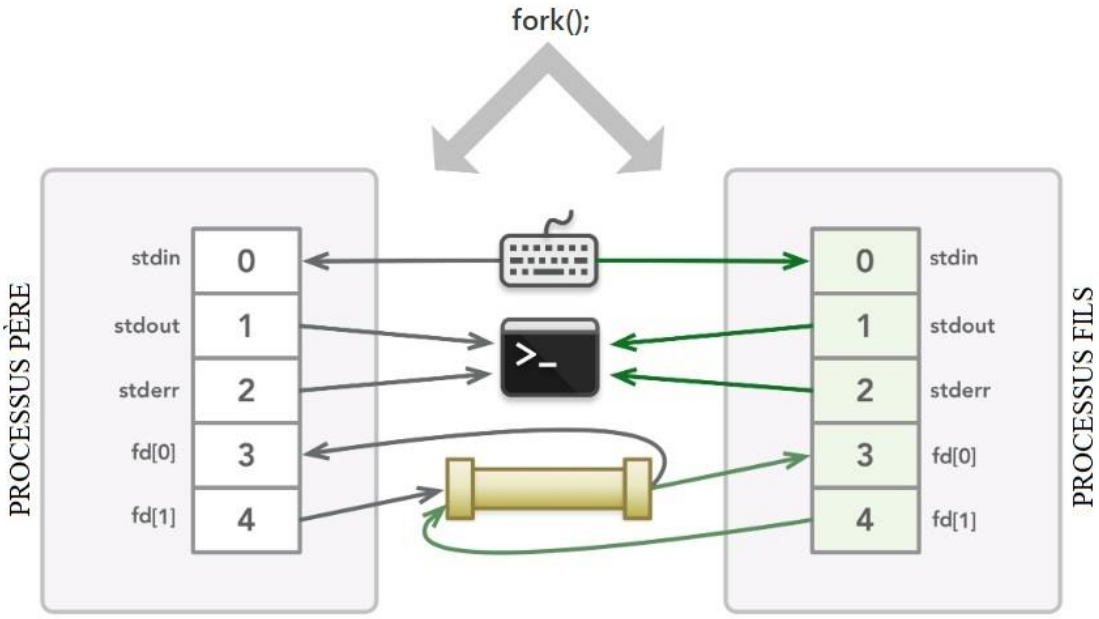
3. Création du fils

Dans le cas d'Unix, l'appel système fork() est la seule véritable façon qui permet de créer des processus fils :

```
#include <unistd.h>

int fork()
```

L'appel système ***fork()*** crée une copie exacte du processus original, comme illustré à la figure en dessus. Schématiquement, cet appel système crée une copie complète du processus qui l'a exécuté. Après exécution de ***fork(),*** il y a deux copies du même processus en mémoire. Le processus qui a exécuté ***fork()*** est considéré comme étant le **processus père** tandis que celui qui a été créé par l'exécution de **fork()** est le **processus fils**.

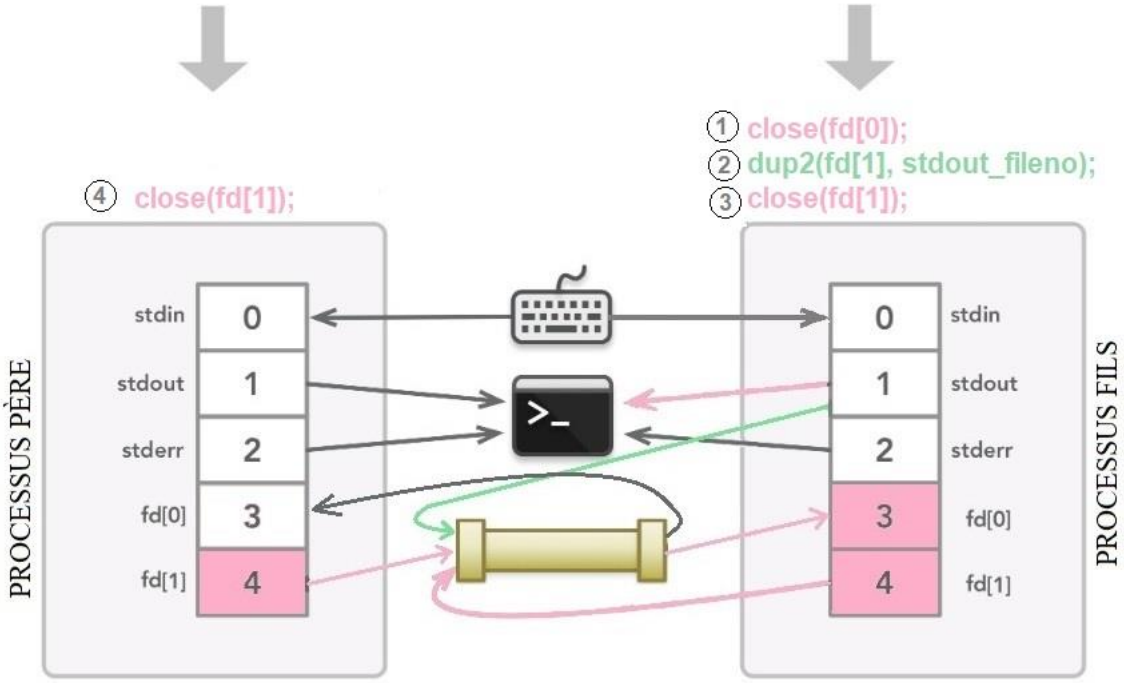


Le *processus père* garde son PID, et le nouveau *processus fils* possède un nouveau PID. Le processus père et le processus fils ont le même code source, mais la valeur retournée par fork permet de savoir si on est dans le processus père ou fils. Ceci permet de faire deux choses différentes dans le processus père et dans le processus fils (en utilisant un if et un else ou un switch).

- L'appel système ***fork()*** retourne la valeur -1 en cas d'erreur. En cas d'erreur, aucun processus n'est créé.
- L'appel système ***fork()*** retourne la valeur 0 dans le processus fils.
- L'appel système ***fork()*** retourne une valeur positive dans le processus père. Cette valeur est l'identifiant du processus fils créé.

4. Redirection de la sortie du fils

Lorsqu'on est dans le fils on fait les étapes suivantes :



① Vu que le fils aura pas besoin de lire dans le tube, on commence par fermer le descripteur[0] « (fd[0]) » du fils.

② Pour copier un descripteur d'un fichier, on fait appeler à la fonction système dup2.

```
#include <unistd.h>

int dup2(int oldfd, int newfd) ;
```

dup2(oldfd, newfd) duplique le descripteur odlfd. Le nouveau descripteur retourné est égal à newfd. Si newfd est utilisé, il est d'abord libéré comme avec ***close()*.** Le nouveau descripteur référence le même fichier que oldfd, utilise le même pointeur de position et a le même mode d'accès (lecture, écriture ...). dup2() retourne la valeur de newfd, et -1 en cas d'erreur.

Pour le travail demandé, il faut lier l'entrée tube[1] du tube à stdout. Par la suite, tout ce qui sort sur le flot de sortie standard stdout du fils entre dans le tube, et pourra être lu par le père par après. On y arrive en écrivant la ligne suivante :

```
dup2(fd[1], STDOUT_FILENO);
```

Comme on peut l'observe sur le schéma au-dessus, le lien entre le stout du fils et le terminal est d'abord coupé en autrement dit, on ferme le stdout du fils puis on relie l'entrée du tube[1] à la sortie du fils.

③ fd[1] du fils est fermé car il est plus utilisé.

④ Dans le parant, on ferme également l'entrée d'écriture car le père va seulement lire dans **le pipe**.

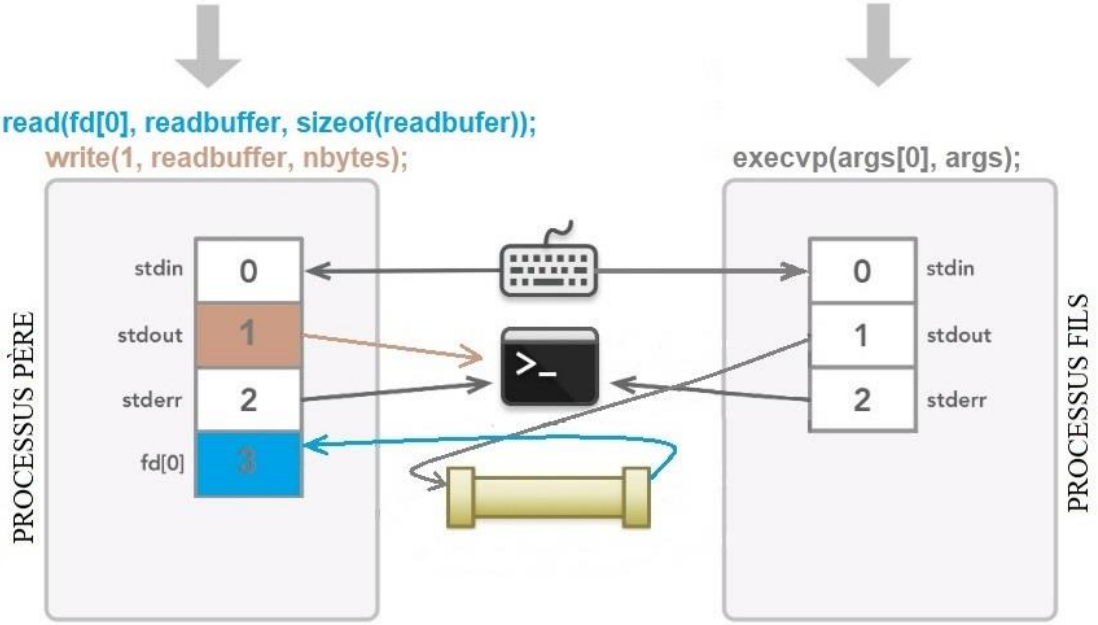
5. Exécution de la commande

Pour l'exécution la commande, on a décidé de passer la commande en paramètres et non via la ligne de commande. Pour faire cela, on a fait appeler à la fonction système ***execvp()***.

Il y a plusieurs fonctions de la famille *exec* mais la fonction qui nous intéresse ici, c'est ***execvp ().*** Ce dernier prend que deux arguments, le premier argument étant la commande à exécuter et le second est un tableau de chaînes de caractères terminé par un élément NULL.

On a choisi d'utiliser ***execvp*** car on ne connait rien du nombre d'arguments. Et donc, on crée un nouveau tableau pour contenir les arguments passé en paramètres et c'est ce tableau-là qu'on passe en second argument **d'*execvp().*** Il faut signaler que le premier élément donné comme argument n'est pas le premier argument mais le nom du programme à exécuter, et donc on s'arrange pour que le nouveau tableau n'a pas le premier élément du tableau argv.

Lors de l'exécution de la commande notre programme fait les étapes suivantes :



① Le nouveau tableau étant crée (args) est passé en paramètre dans la fonction ***execvp()***, si ce dernier arrive à exécuter la commande, les résultats sont envoyés normalement vers la sortie du fils mais cette sortie a été modifié précédemment et maintenant les résultats de la fonction *execvp* vont directement dans le pipe et vont être lu par le parent. Si ***execvp*** arrive pas à exécuter la commande, un message d'erreur est affiché via le descripteur stderr.

② Une fois que les données envoyées par le fils sont bien dans **le pipe**, elles peuvent maintenant être lu et écrite sur le stdout du parent.

Pour lire les données dans le tube, on utilise la fonction système *read ()* et pour écrire sur le stdout du père, on appelle la fonction système ***write().***

```
int read (fd[0], readbuffer, sizeof(readbuffer));
```

- fd[0] c'est là qu'on récupère les info venant du pipe.
- readbuffer est du tableau qu'utilise pour lire les données du pipe.
- 3ème argument c'est taille du tableau utilisé pour lire.

read() renvoie -1 s'il échoue. Sinon, elle renvoie le nombre d'octets lus (0 en fin de fichier). On utilisera ce nombre d'octets lu renvoyé par read pour écrire sur le stout du père.

③ Une fois qu'on a fini de lire sans erreur, on récupère les nombres d'octets lu (nbytes) pour s'en servir lors d'écriture sur le stout du père.

Conclusion :

- ⇨ Sommes-nous parvenu à réaliser l'entièreté du travail ?
- ⇨ Difficultés rencontrées + solutions
- ⇨ Conclusion personnelle

Annexe (code) :

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

#define MAX 1000
#define RED "\x1B[31m"
#define BLU "\x1B[34m"
#define GRN "\x1B[32m"
#define RESET "\x1B[0m"

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

    int nbytes, fd[2];

    /* Buffer used to read bytes from the pipe in the parent. */
    char readbuffer[MAX];

    char *args[argc - 1];
    for (int j = 0; j < argc - 1; j++) {
        args[j] = argv[j + 1];
    }
    /* Null is required at the end of agrs by execvp.*/
    args[argc - 1] = (char *) NULL;

    if (pipe(fd) < 0) fprintf(stderr, "Call to the pipe function failed\n"), exit(EXIT_FAILURE);

    switch (fork()) {
        case -1:
            /* Fork function failed. */
            fprintf(stderr, "Call to fork function failed\n");
            exit(EXIT_FAILURE);
        case 0:                 /* child */
            printf("I'm the child "RED"%d"RESET", my parent is"BLU" %d "RESET" \n", getpid(), getppid());
            printf("I'm the child "RED"%d"RESET", and i'm executing the following command : "GRN"%s"RESET"\n", getpid(),
                   args[0]);

            if (close(fd[0]) == -1) fprintf(stderr, "Couldn't close read end of pipe in child\n"), exit(EXIT_FAILURE);

            if (dup2(fd[1], STDOUT_FILENO) == -1) fprintf(stderr, "dup2 in child failed.\n"), exit(EXIT_FAILURE);

            if (close(fd[1]) == -1) fprintf(stderr, "Couldn't close write end of pipe in child\n"), exit(EXIT_FAILURE);

            execvp(args[0], args);
            /* If the following statement is reached, execvp must have failed. */
            fprintf(stderr, "Call to execvp function in child failed.\n");
            exit(EXIT_FAILURE);
        default:                /* parent */
            if (close(fd[1]) == -1) fprintf(stderr, "Couldn't close write end of pipe in parent\n"), exit(EXIT_FAILURE);

            printf("I'm the parent "BLU"%d"RESET", and i'm printing the result of the command \n", getpid());

            while ((nbytes = read(fd[0], readbuffer, sizeof(readbuffer))) != 0) {
                (nbytes < 0) ? fprintf(stderr, "Call to read function in the parent failed\n"), exit(EXIT_FAILURE)
                             : write(1, readbuffer, nbytes) == -1 ?
                               fprintf(stderr, "Call to write function in parent failed\n"), exit(EXIT_FAILURE)
                                                                  : fprintf(stdout, "End of the program ! \n");
            }
    }
    return 0;
} /* End of main. */

```