Réalisation d'un pipeline en C

Michel Billaud (michel.billaud@laposte.net)

27 septembre 2023

Table des matières

Historique

- 2023-09-27 Correction `demo-exec-dup.c` + typos (merci Damien Simler !)
- 2023-01-15 Version initiale.

1 Objectif

La réalisation d'un mini-*shell* (interprète de commandes) est un projet classique de programmation système en C.

Dans sa version la plus simpliste, un shell est une boucle qui

- affiche une chaîne d'invite (*prompt*),
- lit une commande,
- lance son exécution,
- et recommence.

Les commandes sont des suites de mots : en général le chemin d'accès d'un programme (exécutable), suivi par des options, des arguments…

La réalisation d'un tel programme n'est pas très compliquée.

Là où ça se complique un peu, c'est si on veut que le shell permette d'exécuter des "pipelines" de commandes, c'est-à-dire de lancer plusieurs commandes en redirigeant la sortie de l'une vers l'autre, comme dans

```
ls -l | grep -v ^d | more
```

La difficulté est essentiellement d'utiliser correctement les $n-1$ tuyaux qui interviennent dans un "pipeline" de n commandes.

Nous allons voir ça après quelques rappels.

Attention : pour simplifier la présentation, dans le code ci-dessous on ne vérifie jamais que les appels systèmes ont réussi. Par exemple `fork`, `pipe` etc. peuvent théoriquement échouer, et retourner -1 dans ce cas. En vrai, il faudrait vérifier.

2 Rappels

2.1 Exécuter une commande : `execv`

Le programme suivant

- affiche un message
- fait exécuter la commande `ls /tmp` au moyen de `execv`.

```c
// demo-exec.c

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

int main()
{
    printf("# lancement " __FILE__ "\n");

    char *path        = "/bin/ls";
    char *arguments[] = { "ls", "/tmp", NULL};

    execv(path, arguments);

    perror("échec lancement");
    exit(EXIT_FAILURE);
}
```

La fonction `execv` prend comme paramètres

- le chemin d'accès d'un fichier exécutable,
- une liste de paramètres terminée par le pointeur `NULL`.

Il existe des variantes de cette fonction, notamment `execvp` qui recherche l'exécutable indiqué en premier paramètre dans les répertoires qui figurent dans la variable d'environnement `PATH`. Lire la page de manuel.

2.1.1 Succès de `execv()`

Normalement le fichier indiqué (`/bin/ls`) sera chargé dans la mémoire du processus en remplacement du code de `demo-exec`, et sa fonction `main`, qui a comme prototype `int main(int argc, char **argv)` sera appelée avec

- l'adresse du tableau d'arguments dans `argv` (argument values)
- la valeur 2 dans `argc` (argument count)

Le code qui s'exécute ayant été remplacé, la fin de l'exécution de `/bin/ls` termine le processus : on ne revient pas de `execv`.

2.1.2 Échec de `execv()`

Si le fichier indiqué en premier paramètre est absent, pas accessible, pas exécutable etc. l'appel à `execv` retourne, et les instructions qui suivent sont exécutées.

Après l'appel à `execv`, on trouve donc le code qui gère cet échec.

2.2 Remarques

- La valeur retournée par `execv` en cas d'échec est `-1`, mais peu importe : ce n'est pas la peine de la tester : si on est ressorti d'`execv` c'est que le lancement du programme n'a pas pu se faire.
- ne pas confondre "le lancement a échoué" (dans ce cas `execv` retourne), et "l'exécution du programme lancé a échoué" (le programme lancé s'est effectivement exécuté, et s'est terminé par `exit` en retournant un code non nul).

2.3 Redirection : `dup2`

La plupart des commandes

- lisent des données sur leur **entrée standard**, qui a le descripteur 0 (constante `STDIN_FILENO`),
- écrivent des résultats sur la **sortie standard** (descripteur `STDOUT_FILENO` = 1),
- affichent des messages sur la **sortie d'erreur** (`STDERR_FILENO` = 2).

L'exemple ci-dessous[1] utilise l'appel `dup2` pour que la commande `tr` (lancée avec les paramètres pour convertir les minuscules en majuscules), s'exécute avec son entrée standard redirigée vers un fichier :

[1] correction appel `dup2()` 27 sept 2023 (merci Damien Simler)

```c
// demo-exec-dup.c

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

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main()
{
    printf("# lancement " __FILE__ "\n");

    int file_fd = open("demo-exec-dup.c", O_RDONLY);
    dup2(file_fd, STDIN_FILENO);
    close(file_fd);

    execv("/bin/tr", (char *[]){ "tr", "a-z", "A-Z",  NULL} );

    perror("échec lancement");
    exit(EXIT_FAILURE);

}
```

Remarque : Dans `execv("/bin/tr", (char *[]){"tr", "a-z", "A-Z", NULL});` on utilise un **tableau anonyme** en C.

- Syntaxe du tableau anonyme : le contenu (entre accolades) est précédé par le type du tableau (entre parenthèses).
- But : ici c'est pour exprimer l'appel d'`execv` en une seule ligne, sans botter en touche sur des constantes, pour rendre plus visible la gestion des descripteurs (`dup2`, `close`).

Explications :

- L'appel à la fonction `open` ouvre le fichier en lecture,
- Pour chaque fichier ouvert, il y a une structure de données qui contient un tampon, une indication de l'endroit où on est de la lecture ou de l'écriture, etc.
- Cette structure de données est identifiée par un numéro, le **descripteur de fichier** ouvert. En général le plus petit numéro non utilisé, sans doute 3 ici.
- L'appel `dup2(file_fd, STDIN_FILENO)` fait en sorte que le descripteur `0` conduise au même fichier que le `3` (le descripteur `0` est fermé préalablement).
- Les descripteurs qui sont ouverts le restent lors de l'appel de `execv` : la commande `tr` s'exécute donc avec son entrée standard reliée au fichier de données.
- Préalablement, un `close` de `file_fd` évite une **fuite de descripteur** : on ne veut transmettre que les descripteurs `0`, `1` et `2`.

Ce problème de fuite de descripteur va compliquer la réalisation d'un pipeline.

Exécution :

Le programme affiche le code source en majuscules :

```
# lancement demo-exec-dup.c
// DEMO-EXEC-DUP.C

#INCLUDE <STDIO.H>
#INCLUDE <STDLIB.H>
...
    EXIT(EXIT_FAILURE);

}
```

2.4 Lancer un processus : `fork` + `waitpid`

Comme l'exécution d'une commande par `execv` termine le processus en cours, on va avoir un problème si on veut faire exécuter *plusieurs* commandes.

La solution est de faire exécuter chaque commande par un processus créé à cet effet.

```c
// demo-fork.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include <assert.h>

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

#include <fcntl.h>

void execute_task()
{
    int fd = open("demo-fork.c", O_RDONLY);
    dup2(fd, STDOUT_FILENO);
    close(fd);

    execv("/bin/tr", (char *[]){ "tr", "a-z", "A-Z",  NULL} );
    // en cas de problème pour lancer tr
    perror("erreur lancement commande tr");
    exit(EXIT_FAILURE);
}

void explain_status(int child_status)
{
```

```
    printf("# La commande s'est terminée ");
    if (WIFEXITED(child_status)) {
        printf("par exit(%d)\n", WEXITSTATUS(child_status));
    } else if (WIFSIGNALED(child_status)) {
        printf("par la réception du signal %d\n", WTERMSIG(child_status));
    } else {
        printf("pour une raison x ou y.\n");
    }
}

int main(int argc, char *argv[])
{
    pid_t child_pid = fork();
    if (child_pid == 0) {
        execute_task();
        assert("on ne revient jamais ici" && false);
    }
    int child_status;
    waitpid(child_pid, &child_status, 0);

    explain_status(child_status);
    exit(EXIT_SUCCESS);
}
```

L'exécution montre ceci :

```
// DEMO-FORK.C

#INCLUDE <STDIO.H>
#INCLUDE <STDLIB.H>
....
    EXPLAIN_STATUS(CHILD_STATUS);
    EXIT(EXIT_SUCCESS);
}
# La commande s'est terminée par exit(0)
```

Explications brèves (qui ne remplacent pas un cours)

1. L'appel à `fork()` démarre un nouveau processus (dit "fils") qui est une copie du processus en cours d'exécution (son "père").
   - les deux processus partagent les ressources (descripteurs de fichiers),
   - `fork()` retourne au père l'identifiant du processus fils, et 0 à celui-ci.

Le processus fils appelle donc la fonction `execute_task()`, tandis que le processus père qui a reçu l'identifiant non nul de son fils ne rentre pas dans le corps du `if`.

2. La fonction `execute_task()` - exécutée par le fils - lance la commande `/bin/tr` comme dans l'exemple précédent. On ne revient jamais de cette fonction qui lance `/bin/tr` par `execv`, ou `exit(EXIT_FAILURE)` en cas d'échec.

3. En même temps (à peu près) le processus père

- exécute `waitpid` qui le bloque en attendant la fin de l'exécution du fils,
- affiche ensuite des messages et se termine.

4. Le second paramètre de `waitpid` est l'adresse d'un entier qui contiendra les informations sur la fin du processus fils. Dans `explain_child_status`, diverses macros permettent
   - de savoir si il s'est terminé par `exit()` ou en recevant un signal qui a provoqué sa fin,
   - de connaître le code de retour dans le premier cas, ou le numéro du signal.

2.5 Faire communiquer deux commandes par `pipe`

Un "tuyau" est un pseudo-fichier géré par le système d'exploitation. C'est un tampon en mémoire dans lequel on peut écrire et lire. Il sert à la communication entre processus issus d'un même père. Les données sont lues dans l'ordre où elles ont été écrites dans le tuyau.

Un tuyau a une capacité limitée. Un processus qui veut écrire plus de données dans le tuyau qu'il ne peut en contenir sera bloqué en attendant qu'il y ait suffisamment de place.

Le tuyau est créé par un appel à la commande `pipe` qui retourne - dans un tableau donné en paramètre - deux descripteurs : celui qui sert à lire dans le tuyau, et celui qui sert à y écrire.

Plusieurs processus peuvent lire et/ou écrire dans un même tuyau, pourvu qu'ils disposent d'un descripteur ouvert qui permet l'opération.

Important : En lecture, on atteint la "**fin de fichier**" quand il n'y a plus de données disponibles dans le tuyau, **et que tous les descripteurs d'écriture ont été fermés**.

Pour chaque processus, il faudra donc faire très attention à bien refermer tous les descripteurs (surtout d'écriture) dont on ne se sert pas. Sinon un processus en lecture pourra rester bloqué indéfiniment, en attente de données que personne ne lui enverra.

L'exemple qui suit est equivalent au lancement de la commande shell :

```
date | tr a-z A-Z
```

qui affiche la date mise en majuscules

```
SAM. 14 JANV. 2023 18:50:22 CET
```

Voici le source :

```
// demo-pipe.c
// demo-pipe.c

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

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

```c
#include <sys/stat.h>
#include <sys/wait.h>

#include <fcntl.h>

int main(int argc, char *argv[])
{
    int pipe_fd[2];

    pipe(pipe_fd);

    pid_t date_pid = fork();
    if (date_pid == 0) {
        // code exécuté par le premier processus fils (date)
        dup2(pipe_fd[1], STDOUT_FILENO);
        close(pipe_fd[0]);
        close(pipe_fd[1]);
        execv("/bin/date", (char *[]){"date", NULL});
        perror("échec lancement de date");
        exit(EXIT_FAILURE);
    }

    pid_t tr_pid = fork();
    if (tr_pid == 0) {
        // code exécuté par le second processus fils (tr)
        dup2(pipe_fd[0], STDIN_FILENO);
        close(pipe_fd[0]);
        close(pipe_fd[1]);
        execv("/bin/tr", (char *[]){"tr", "a-z", "A-Z", NULL});
        perror("échec lancement de tr");
        exit(EXIT_FAILURE);
    }

    close(pipe_fd[0]);
    close(pipe_fd[1]);
    waitpid(date_pid, NULL, 0);
    waitpid(tr_pid, NULL, 0);

    exit(EXIT_SUCCESS);
}
```

Explications :

- L'appel système `pipe(pipe_fd)` crée un tuyau et place les descripteurs dans le tableau.
- Le premier fils hérite des deux descripteurs
  - il duplique le descripteur d'écriture, pour que la sortie de la commande `date` se fasse dans le tuyau ;
  - il ferme ensuite les descripteurs du tuyau.
- Le second fils hérite aussi des deux descripteurs
  - il duplique le descripteur de lecture, pour que la commande `tr` prenne

son entrée dans le tuyau ;
  - il ferme ensuite les descripteurs du tuyau.
- une fois les fils lancés, le processus père
  - ferme les descripteurs du tuyau (il n'en n'a plus l'usage),
  - attend la fin de l'exécution des deux fils.

Remarque : Comme le descripteur d'écriture `pipe_fd[1]` n'est utilisé que par le premier processus fils, le processus père pourrait le refermer immédiatement après le lancement du premier fils, ce qui dispenserait d'avoir à le faire dans aussi le code du second. Mais on a privilégié ici la simplicité du code.

2.5.1 Résumé : les mécanismes Unix utilisés

- `fork()` crée un processus "fils" qui est une copie de celui qui l'appelle (= père). Il retourne 0 au fils, et le numéro du fils au père. Les descripteurs ouverts sont partagés.
- `exit()` termine le processus qui l'appelle.
- `waitpid()` bloque un processus en attente de la fin d'un autre dont on donne le numéro.
- `pipe()` crée un tuyau, et retourne dans un tableau une paire de descripteurs vers les extrémités qui servent à y lire et y écrire.
- `dup2()` duplique un descripteur, ce qui permet de rediriger une entrée ou une sortie vers un fichier ou un pipe.
- `close()` ferme un descripteur.
- `execv()` remplace le processus courant par l'exécution d'un programme en lui transmettant des arguments, et en partageant les descripteurs ouverts. L'exit du programme terminera le processus. En cas d'échec de lancement (fichier absent, non exécutable, etc) l'appelant continue.

3 Pipeline : analyse sur un exemple

Quand on aborde un problème un peu compliqué, il est conseillé de commencer par regarder un exemple concret, petit mais significatif. À partir de là, on pourra construire plus facilement une solution générale.

Considérons donc un exemple de pipeline qui met en oeuvre 4 programmes :

```
A | B | C | D
```

pourquoi 4 ?

- parce que pour évoquer un pipeline, il faut au moins considérer 2 commandes ;
- parce que la première et la dernière commandes sont des cas particuliers qui respectivement lisent dans l'entrée standard, et écrivent sur la sortie standard, alors que les autres opérations se font sur un pipe.
- avec 2 commandes "intermédiaires" B et C, on aura des chances d'inférer ce qu'il faut faire avec un nombre *quelconque* de commandes intermédiaires.

3.1 Idée de base : 4 processus fils, 3 tuyaux

L'idée de base sera de lancer un processus pour chaque commande, et d'utiliser trois tuyaux T1, T2, T3 pour les faire communiquer

```
lancer processus fils:
    exécuter A, qui
    - lit sur l'entrée standard
    - écrit dans T1

lancer processus fils:
    exécuter B, qui
    - lit dans T1
    - écrit dans T2

lancer processus fils:
    exécuter C, qui
    - lit dans T2
    - écrit dans T3

lancer processus fils:
    exécuter D, qui
    - lit dans T4
    - écrit sur la sortie standard

attendre la fin des processus fils.
```

3.2 Créer et fermer les descripteurs

Détaillons le **lancement du premier** :

1. Le tuyau T1 doit exister avant le lancement du premier processus fils , puisque le tuyau doit être visible par le second processus.
2. Le processus A ne lit pas dans T1, le processus fils peut fermer le bout qui sert à l'écriture.
3. les autres processus ne doivent pas écrire dans ce tuyau : une fois le fils lancé, on ferme le bout qui sert l'écriture dans le tuyau.

```
créer tuyau T1                          // 1
lancer processus fils:
    fermer T1[1]                        // 2
    exécuter A, qui
    - lit sur l'entrée standard
    - écrit dans T1[1]
fermer T1[1] et entrée standard
```

Le second processus doit lire dans T1 et écrire dans T2.

1. Il faut donc créer T2 avant de lancer la commande B,
2. la commande B2 ne lit pas dans T2, le processus fils ferme cette extrémité
3. les processus suivants ne lisent pas dans T1, et n'écrivent pas dans T2.

```
créer tuyau T2                          // 1
```

```
lancer processus fils:
    fermer T2[1]                        // 2
    exécuter B, qui
    - lit sur T1[0]
    - écrit dans T2[1]
fermer T1[0] et T2[1]                   // 3
```

La situation est similaire pour le troisième

```
créer tuyau T3                          // 1
lancer processus fils:
    fermer T3[1]                        // 2
    exécuter C, qui
    - lit sur T2[0]
    - écrit dans T3[1]
fermer T2[0] et T3[1]                   // 3
```

pour la dernière commande, on ne crée pas de tuyau puisqu'on écrit sur la sortie standard.

```
lancer processus fils:
    exécuter D, qui
    - lit sur T3[0]
    - écrit sur la sortie standard
fermer T3[0]
```

3.3 En termes de descripteurs

Nous allons nous intéresser aux descripteurs plutot qu'aux tableaux, en passant par des variables, on écrit la création d'un tuyau sous la forme `créer tuyau (sortie_tuyau, entrée_tuyau)` en indiquant les deux variables qui contiennent les descripteurs d'écriture et de lecture.

Une autre variable désigne le descripteur utilisé en lecture par le prochain processus.

Au départ, c'est l'entrée standard :

```
entrée = entrée_standard

créer tuyau (sortie_tuyau, entrée_tuyau)
lancer processus fils:
    fermer entrée_tuyau
    exécuter A, qui
    - lit sur entrée
    - écrit dans sortie_tuyau
fermer sortie_tuyau et entrée
```

Le pseudo-code pour les processus B et C n'est pas très différent :

```
entrée = entrée_tuyau

créer tuyau (sortie_tuyau, entrée_tuyau)
lancer processus fils:
```

```
    fermer entrée_tuyau
    exécuter B, qui
    - lit sur entrée
    - écrit dans sortie_tuyau
fermer sortie_tuyau et entrée

entrée = entrée_tuyau

créer tuyau (sortie_tuyau, entrée_tuyau)
lancer processus fils:
    fermer entrée_tuyau
    exécuter C, qui
    - lit sur entrée
    - écrit dans sortie_tuyau
fermer sortie_tuyau et entrée
```

Pour le dernier processus fils, on ne crée pas de tuyau :

```
entrée = entrée_tuyau

lancer processus fils:
    fermer entrée_tuyau
    exécuter D, qui
    - lit sur entrée
    - écrit dans sortie_standard
fermer entrée
```

3.4 Programmation de l'exemple en C

Ici on réalise le pipeline

```
ls -l | grep ^- | cat | wc -l
```

qui affiche le nombre de fichiers présents dans le répertoire courant. (Le `cat` est inutile, il sert juste à avoir 4 processus !).

```c
// demo-pipeline.c

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <assert.h>

int main()
{
    printf("Exécution " __FILE__ "\n");
    printf("Nombre de fichiers dans le répertoire =\n");
```

```
int input_fd = STDIN_FILENO;
int pipe_fd[2];

pipe(pipe_fd);                    // création pipe T1

pid_t a_pid = fork();
if (a_pid == 0) {
    // le processus A écrit dans T1
    dup2(pipe_fd[1], STDOUT_FILENO);
    close(pipe_fd[0]);
    close(pipe_fd[1]);
    execv("/bin/ls", (char *[]) {
        "ls", "-l", NULL
    });
    assert("lancement premier fils" && false);
}
close(pipe_fd[1]);          // fermeture desc. écriture de T1

input_fd = pipe_fd[0];      // sauvegarde descripteur lecture T1
pipe(pipe_fd);              // création pipe T2

pid_t b_pid = fork();
if (b_pid == 0) {
    // le processus B lit dans T1, écrit dans T2
    dup2(input_fd, STDIN_FILENO);
    dup2(pipe_fd[1], STDOUT_FILENO);
    close(input_fd);
    close(pipe_fd[0]);
    close(pipe_fd[1]);
    execv("/bin/grep", (char *[]) {
        "grep", "^-", NULL
    });
    assert("lancement second fils" && false);
}
close(input_fd);             // descripteur lecture T1
close(pipe_fd[1]);           // descripteur écriture T2

input_fd = pipe_fd[0];       // sauvegarde descripteur lecture T2
pipe(pipe_fd);               // création pipe T3

pid_t c_pid = fork();
if (c_pid == 0) {
    // le processus C lit dans T2, écrit dans T3
    dup2(input_fd, STDIN_FILENO);
    dup2(pipe_fd[1], STDOUT_FILENO);
    close(input_fd);         // desc lecture T2
    close(pipe_fd[0]);       // desc lecture T3
    close(pipe_fd[1]);       // desc écriture T3
    execv("/bin/cat", (char *[]) {
        "cat", NULL
```

```
        });
        assert("lancement troisième fils" && false);
    }
    close(input_fd);        // desc lecture  T2
    close(pipe_fd[1]);      // desc écriture T3

    input_fd = pipe_fd[0];  // sauvegarde descripteur lecture T3


    pid_t d_pid = fork();
    if (d_pid == 0) {
        // le processus C lit dans T3
        dup2(input_fd, STDIN_FILENO);
        close(input_fd);         // desc lecture T3
        execv("/bin/wc", (char *[]) {
            "wc", "-l", NULL
        });
        assert("lancement quatrième fils" && false);
    }

    close(input_fd);

    waitpid(a_pid, NULL, 0);
    waitpid(b_pid, NULL, 0);
    waitpid(c_pid, NULL, 0);
    waitpid(d_pid, NULL, 0);

    printf("# fin\n");
    return EXIT_SUCCESS;
}
```

4 Pipeline : cas général

4.1 Principe

L'exemple ci-dessus nous permet de voir les actions à effectuer pour chacun des processus fils, avec les particularités du premier et du dernier.

Le processus père :

1. Avant de créer un processus fils (sauf le dernier), le processus père réserve un tuyau pour communiquer avec le processus suivant.
2. Après avoir lancé un processus fils (sauf le premier), ferme `input_fd`.
3. Après avoir lancé un processus fils (sauf le dernier) :
   - ferme le descripteur d'écriture du tuyau
   - sauve le descripteur de lecture du tuyau dans `input_fd`

Chaque processus fils :

1. redirige (sauf le premier) son entrée vers `input_fd`, et ferme `input_fd`

2. redirige sa sortie (sauf le dernier) vers le tuyau, et ferme les deux descripteurs du pipe.

Il est assez facile de vérifier que ces règles fonctionnent même si il n'y a qu'un seul processus fils (le premier est aussi le dernier) ou deux (pas de processus intermédiaires).

4.2 Un code plus général

Pour avoir un traitement plus général, nous allons définir une fonction

```
void execute_pipeline(struct Pipeline *pipeline);
```

qui agit sur une structure qui représente un pipeline constitué d'étapes

```
struct Step {
    char *pathname;
    char **argv;
};

struct Pipeline {
    int nb_steps;
    struct Step *steps;
};
```

La fonction `main` de l'exemple précédent se ramènerait à

```
int main()
{
    struct Pipeline pipeline = {
        .nb_steps = 4,
        .steps = (struct Step [])
        {
            {
                .pathname = "/bin/ls",
                .argv = (char *[]) {
                    "ls", "-l", NULL
                }
            }, {
                .pathname = "/bin/grep",
                .argv = (char *[])
                {
                    "grep", "^-", NULL
                }
            }, {
                .pathname = "/bin/cat",
                .argv = (char *[])
                {
                    "cat", NULL
                }
            }, {
                .pathname = "/bin/wc",
                .argv = (char *[])
```

```
                {
                    "wc", "-l", NULL
                }
            },
        }
    };

    printf("Exécution " __FILE__ "\n");
    printf("Nombre de fichiers dans le répertoire =\n");

    execute_pipeline(& pipeline);

    printf("# fin\n");
    return EXIT_SUCCESS;
}
```

La fonction `execute_pipeline` boucle sur les éléments du pipeline, en tenant compte des cas particuliers du premier et du dernier :

```
void execute_pipeline(const struct Pipeline *pipeline)
{
    const int first = 0,
              last = pipeline->nb_steps - 1;

    int input_fd = STDIN_FILENO;

    for (int i = first; i <= last; i++) {
        int pipe_fd[2];
        if (i != last) {
            pipe(pipe_fd);
        }
        if (fork() == 0) {
            // exécution d'un processus fils
            if (i != first) {
                dup2(input_fd, STDIN_FILENO);
                close(input_fd);
            }
            if (i != last) {
                dup2(pipe_fd[1], STDOUT_FILENO);
                close(pipe_fd[0]);
                close(pipe_fd[1]);
            }
            execv(pipeline->steps[i].pathname,
                  pipeline->steps[i].argv);
            assert("lancement fils" && false);
        }
        if (i != first) {
            close(input_fd);
        }
        if (i != last) {
            close(pipe_fd[1]);
```

```
            input_fd = pipe_fd[0];
        }
    }

    for (int i = 0; i <= last; i++) {
        wait(NULL);
    }
}
```

A la fin, à la place de `waitpid`, on emploie `wait` qui attend la fin d'un processus fils quelconque, sans devoir préciser son identifiant.

5 Exercice : une solution récursive

On peut remarquer qu'un pipeline est composé

- soit d'une commande seule,
- soit d'une commande qui envoie sa sortie standard dans un pipe qui est lu par le pipeline des commandes suivantes.

Ca peut donner l'idée d'une solution récursive, une fonction qui prend comme paramètres :

- une liste de commandes,
- des descripteurs pour l'entrée et la sortie du pipeline.

Schématiquement :

```
pour exécuter une liste de commandes:
    si la liste contient une seule commande:
        lancer un processus fils qui:
            exécute la commande
        attendre la fin de la commande
    sinon:
        créer un tuyau
        lancer un processus fils qui:
             redirige la sortie vers le tuyau
             exécute la première commande
        rediriger l'entrée vers le tuyau
        exécuter le reste des commandes
        attendre la fin de la première commande
```

Comme souvent, la version récursive est plus simple que la version itérative !

6 Remarques finales

Faire fonctionner un pipeline de commandes n'est pas un problème très compliqué, mais dont la solution comporte pas mal de détails techniques à manipuler avec précision.

Pour ajouter des pipelines dans un interprète de commandes (shell), il est donc fortement recommandé d'étudier **à part** la manière de programmer leur

exécution. Sinon, on va combiner les problèmes d'exécution du pipeline aux autres qui se posent déjà dans le projet (analyse syntaxique des commandes, par exemple), et ça ne fera pas gagner de temps.

Cette remarque vaut évidemment pour tous les projets : séparer autant que possible les problèmes pour les étudier, avant d'essayer de les intégrer dans le projet.

Un autre point, la méthode de travail : avant de se lancer dans la réalisation d'un mécanisme général (ici exécution d'un pipeline de N commandes quelconques), **regarder en détail un ou plusieurs cas concrets** (ici le pipeline de 4 commandes) pour bien voir les problèmes qui se posent. C'est comme ça qu'on acquiert les éléments de compréhension que l'on peut **appliquer ensuite à la recherche d'une solution dans le cas général**.

Enfin : l'utilisation de pseudo-code permet de fixer sur papier (ou dans un fichier) les grandes lignes du code qu'on envisage d'écrire. Et donc de se concentrer ensuite sur les divers détails, sans perdre la vue d'ensemble qu'on n'arrivera pas à garder en mémoire dès qu'il y en un certain nombre, ou qu'ils requièrent une forte attention (limite de la charge cognitive).