

COURS PIPEX

42

Introduction

Pipex est un programme en C qui réplique le comportement d'un pipe en ligne de commande Unix. Il permet d'exécuter deux commandes en les connectant via un tube de communication inter-processus.

Commande équivalente en shell :

```
< infile cmd1 | cmd2 > outfile
```

Objectif :

- Lire un fichier `infile`
- Exécuter `cmd1`, transmettre sa sortie à `cmd2`
- Écrire le résultat final dans `outfile`

Ce projet met en avant l'utilisation de plusieurs concepts essentiels du système Unix : la gestion des processus (`fork()`), la redirection d'entrée/sortie (`dup2()`), la gestion des pipes (`pipe()`) et l'exécution de commandes (`execve()`).

Gestion des erreurs

L'utilisation de `perror()` permet d'afficher des messages d'erreur plus détaillés et informatifs. Il est crucial de gérer les erreurs pour éviter des comportements imprévisibles. Par exemple :

- Échec de `pipe()` : Si la création du tube échoue, cela signifie qu'aucun canal de communication ne peut être établi.
- Échec de `fork()` : Indique que le système n'a pas pu allouer un nouveau processus, souvent dû à une surcharge du système.
- Erreurs de redirection `dup2()` : Si la redirection d'entrée ou de sortie échoue, le programme ne fonctionnera pas comme prévu.

Exemple d'affichage d'erreur :

```
1. if (pipe(pipefd) == -1) {  
2.     perror("Pipe error");  
3.     exit(EXIT_FAILURE);  
}
```

- Cela aide à comprendre précisément quelle erreur s'est produite et où.

Gestion de la mémoire

La fonction `free_split()` évite les fuites mémoire en libérant les chaînes allouées dynamiquement. Lors de l'exécution de commandes, nous allouons dynamiquement des tableaux de chaînes pour stocker les arguments. Il est impératif de libérer cette mémoire après usage pour éviter les fuites. Exemple en bas

- Sans cette fonction, des fuites de mémoire peuvent s'accumuler, ce qui peut rendre le programme instable et gourmand en ressources

Gestion des chemins d'accès

Les fonctions `ft_pwd_path()` et `ft_path()` assurent la localisation correcte des commandes et construire correctement les chemins des commandes en gérant :

- Les chemins relatifs (ex : `./a.out`) et absolus (ex: `/usr/bin/ls`).
- La recherche dans `$PATH`

Le système cherche d'abord si la commande est spécifiée avec un **chemin absolu** (ex: `/bin/ls`). Si ce n'est pas le cas, il vérifie dans *la variable d'environnement* `$PATH`.

Exemple de recherche du chemin d'une commande plus loin.

Dans l'environnement de bureau GNOME, « `gnome-session` » est le processus parent de tous les processus s'exécutant dans cet environnement. Ceci constitue (avec le principe d'héritage) le point clé qui nous permet d'influencer le comportement de notre environnement de bureau grâce aux variables d'environnement. L'équivalent pour KDE est « `kde-session` ».

Exécution des commandes

La fonction `ft_get_cmd()` assure correctement :

- le Découpe la commande et ses arguments
- la Recherche du chemin correct dans `$PATH`.
- Exécution de la commande avec `execve()`

Lorsqu'une commande est exécutée, `execve()` remplace l'image du processus en cours par celle de la nouvelle commande, ce qui signifie que si `execve()` réussit, aucune instruction suivante n'est exécutée.

- Cela garantit que chaque commande saisie par l'utilisateur est bien interprétée et exécutée

Gestion des processus

parent/enfant :

Les fonctions `ft_parent_process()` et `ft_child_process()` distinguent les rôles parent/enfant :

Elles gèrent la redirection correcte des entrées et sorties standard (`stdin` et `stdout`) avec `dup2()`, ce qui permet :

- Redirection avec `dup2()`
- Création des processus avec `fork()`
- *S'assurer que `cmd1` écrit bien dans le pipe.*
- *Que `cmd2` lit bien depuis le pipe.*

`fork()` crée un nouveau processus qui est une copie de son parent. Chaque appel à `fork()` double le nombre de processus en cours d'exécution.

Gestion des pipes

Un pipe est une structure de données permettant une communication unidirectionnelle entre deux processus.

- `pipefd[0]` : extrémité de lecture
- `pipefd[1]` : extrémité d'écriture

Le processus qui écrit dans le pipe doit fermer `pipefd[0]`, et celui qui lit doit fermer `pipefd[1]`.

Les descripteurs de fichiers des pipes sont fermés dans les processus adéquats pour :

- Éviter les fuites de ressources.
- Assurer un flux de données correct.
- Prévenir les blocages (deadlocks), qui pourraient survenir si un processus attend une entrée qui ne viendra jamais.

Attente des processus enfants

`waitpid()` permet au parent d'attendre que tous les processus enfants terminent avant de continuer ou de se fermer. Cela évite que le programme ne termine avant que toutes les commandes aient fini de s'exécuter ou d'empêcher la création de processus zombies.

1. **Création du pipe** : `pipe(pipex->pipefd);`
2. **Premier processus (cmd1)** :
 - Redirige `stdin` vers `infile`
 - Redirige `stdout` vers `pipe`
 - Exécute `cmd1`
3. **Second processus (cmd2)** :
 - Redirige `stdin` vers `pipe`
 - Redirige `stdout` vers `outfile`
 - Exécute `cmd2`
4. **Fermeture des pipes** et attente des processus enfants.

Explication du fonctionnement

Le Processus Parent (**pipex**) démarre 🚀

- Il crée un **pipe** (`pipe(pipex->pipefd)`).
- Il lance ensuite deux **processus enfants** (`fork()`).

Création du Pipe 🔗

- Le pipe a **deux extrémités** :
 - `pipefd[1]` : Permet d'écrire dans le pipe (utilisé par `cmd1`).
 - `pipefd[0]` : Permet de lire depuis le pipe (utilisé par `cmd2`).

Le premier processus enfant (**cmd1**) est lancé 🧑

- Il redirige son **stdin** vers **infile**.
- Il redirige son **stdout** vers **pipefd[1]** (`dup2(pipefd[1], STDOUT_FILENO)`).
- Il exécute `cmd1`, qui écrit dans le pipe.

Le second processus enfant (**cmd2**) est lancé 🧑

- Il redirige son **stdin** vers **pipefd[0]** (`dup2(pipefd[0], STDIN_FILENO)`).
- Il redirige son **stdout** vers **outfile** (`dup2(exec->outfile_fd, STDOUT_FILENO)`).
- Il exécute `cmd2`, qui lit les données du pipe et écrit le résultat dans **outfile**.

Attente des Processus Enfants (`waitpid()`) ⌚

- Après avoir lancé `cmd1` et `cmd2`, le **processus parent ferme le pipe**.
- Il **attend** la fin des **deux processus enfants** (`waitpid(exec.pid1, NULL, 0)` puis `waitpid(exec.pid2, &status, 0)`).
- Quand les enfants terminent, le parent quitte.

Le fichier `outfile` est généré avec les résultats 📄

- `cmd2` écrit son **résultat final** dans `outfile`.
- Le programme Pipex se termine.

📝 Remarque supplémentaire :

`pipex.h` : Assurez-vous que ce fichier d'en-tête contient bien toutes les bibliothèques nécessaires, notamment :

```
#include <unistd.h> // Gestion des processus et redirections

#include <stdlib.h> // Gestion de la mémoire

#include <stdio.h> // Affichage et gestion des erreurs

#include <string.h> // Manipulation de chaînes de caractères

#include <sys/wait.h> // Attente des processus enfants

#include <fcntl.h> // Manipulation des fichiers (open, close)
```

- *Ces bibliothèques sont essentielles pour assurer le bon fonctionnement de Pipex, notamment pour la gestion des fichiers, des processus, et des entrées/sorties.*

Analyse du code

📌 1. Fichier `main.c` (Point d'entrée du programme) 📄

```

int main(int argc, char **argv, char **envlp)

{

    t_pipex pipex;

    if (argc != 5)

    {

        ft_putstr_fd("Usage: ./pipex infile cmd1 cmd2 outfile\n", 2);

        exit(EXIT_FAILURE);

    }

    init_pipex(&pipex, envlp, argv[4]);

    run_pipex(&pipex, argv);

    return (EXIT_SUCCESS);

}

```

Explication :

Le programme **vérifie** si l'utilisateur a fourni **quatre arguments** (**infile**, **cmd1**, **cmd2**, **outfile**).

Il **initialise la structure **pipex**** via **init_pipex()**.

Il **exécute le pipeline** en appelant **run_pipex()**.

Lien avec d'autres fonctions :

init_pipex() : Remplit la structure **pipex** avec l'environnement et le nom du fichier de sortie.

`run_pipex()` : Lance l'exécution des processus et des pipes.

2. `init_pipex()` (Initialisation de pipex)

```
void    init_pipex(t_pipex *pipex, char **envlp, char *outfile_name)
{

    pipex->envlp = envlp;

    pipex->outfile_name = outfile_name;

}
```

Explication :

- Cette fonction **remplit** la structure `pipex` avec :
 - L'environnement (`envlp`).
 - Le nom du fichier de sortie (`outfile_name`).

Lien avec d'autres fonctions :

- `pipex` est ensuite utilisé dans `run_pipex()` pour gérer les fichiers et les processus.

3. `run_pipex()` (Gestion du pipeline)

```
void    run_pipex(t_pipex *pipex, char **argv)
{

    t_exec  exec;

    int     status;
```



```
ft_memset(&exec, 0, sizeof(t_exec));

handle_input(&exec, argv[1]);

if (pipe(pipeex->pipefd) == -1)

    error_exit("Pipe error", NULL);

exec.pid1 = fork();

if (exec.pid1 == 0)

    first_child(pipeex, &exec, argv[2]);

close(pipeex->pipefd[1]);

exec.pid2 = fork();

if (exec.pid2 == 0)

    second_child(pipeex, &exec, argv[3]);

close(pipeex->pipefd[0]);

close(exec.infile_fd);

waitpid(exec.pid1, NULL, 0);

waitpid(exec.pid2, &status, 0);

if (WIFEXITED(status))

    exit(WEXITSTATUS(status));

exit(EXIT_FAILURE);
```

```
}
```

Explication :

1. Initialise la structure **exec** avec `ft_memset()`.
2. Ouvre le fichier **infile** avec `handle_input()`.
3. Crée un pipe (`pipe(pipe->pipefd)`) pour connecter **cmd1** et **cmd2**.
4. Crée le premier processus (**cmd1**) avec `fork()` et `first_child()`.
5. Crée le second processus (**cmd2**) avec `fork()` et `second_child()`.
6. Ferme les pipes dans le parent.
7. Attend que les deux processus terminent (`waitpid()`).

Lien avec d'autres fonctions :

- `handle_input()` : Ouvre **infile**.
- `first_child()` : Exécute **cmd1** en écrivant dans le pipe.
- `second_child()` : Exécute **cmd2** en lisant depuis le pipe.

4. **first_child()** (Exécution de la première commande)

```
void    first_child(t_pipex *pipex, t_exec *exec, char *cmd)
{

    close(pipex->pipefd[0]);

    if (dup2(exec->infile_fd, STDIN_FILENO) == -1)

        error_exit("Error: Dup2 failed (input)", NULL);

    close(exec->infile_fd);

    if (dup2(pipex->pipefd[1], STDOUT_FILENO) == -1)

        error_exit("Error: Dup2 failed (output)", NULL);

    close(pipex->pipefd[1]);
```

```
execute_command(cmd, pipex->envlp);  
  
}
```

Explication :

1. Ferme l'extrémité de lecture du pipe (`pipefd[0]`).
2. **Redirige `stdin` vers `infile`** (`dup2()`).
3. **Redirige `stdout` vers `pipefd[1]`** (pour envoyer les données au second processus).
4. **Exécute `cmd1`** avec `execute_command()`.

Lien avec d'autres fonctions :

- `execute_command()` : Exécute la commande avec `execve()`.

5. `second_child()` (Exécution d'une commande avec `execve`)

```
void    second_child(t_pipex *pipex, t_exec *exec, char *cmd)  
{  
  
    t_child child_data;  
  
    exec->outfile_fd = open(pipex->outfile_name, O_WRONLY | O_CREAT |  
O_TRUNC, 0644);  
  
    if (exec->outfile_fd == -1)  
  
        error_exit("Error: Outfile", NULL);  
  
    child_data.input_fd = pipex->pipefd[0];  
  
    child_data.output_fd = exec->outfile_fd;  
  
    child_data.cmd = cmd;
```

```

child_process(pipeex, exec, &child_data);

close(exec->outfile_fd);

close(pipeex->pipefd[0]);

close(pipeex->pipefd[1]);

}

```

Explication :

1. Ouvre **outfile** pour la sortie finale.
2. Prépare les **descripteurs de fichiers** pour **child_process()**.
3. Exécute **cmd2** en redirigeant son entrée depuis le pipe et sa sortie vers **outfile**.

Lien avec d'autres fonctions :

- **child_process()** : Exécute la commande.
- **execute_command()** : Exécute le programme.

6. **execute_command()** (Lance une commande avec **execve**)

Code extrait :

```

void    execute_command(char *cmd, char **envp)

{

    char    **cmd_args;

    char    *path;

    cmd_args = split_command(cmd);

    if (!cmd_args || !cmd_args[0])

```

```

        clean_exit(cmd_args, "", "command not found", 127);

    if (ft_strchr(cmd_args[0], '/'))

        handle_absolute_path(cmd_args, envp);

    path = find_command_path(cmd_args[0], envp);

    if (!path)

        clean_exit(cmd_args, cmd_args[0], "command not found", 127);

    execve(path, cmd_args, envp);

    free(path);

    clean_exit(cmd_args, cmd_args[0], "execve failed", EXIT_FAILURE);
}

```

Explication :

1. **Découpe la commande en arguments** avec `split_command()`.
2. **Cherche le chemin de la commande** (`find_command_path()`).
3. **Exécute la commande** avec `execve()`.

Analyse du fonctionnement de `execute_command`

La fonction `execute_command` est essentielle pour exécuter une commande donnée (`cmd`) en cherchant son chemin dans le `PATH` et en lançant `execve`.

Elle fonctionne en plusieurs étapes :

1. **Validation de la commande** : Vérifie que la commande n'est pas vide.
2. **Découpage (`split_command`)** : Décompose `cmd` en arguments séparés.
3. **Vérification d'un chemin absolu ou relatif** : Si la commande contient `/`, elle est exécutée directement.
4. **Recherche du chemin (`find_command_path`)** : Si ce n'est pas un chemin absolu, on cherche dans le `PATH`.
5. **Exécution avec `execve`** : Lance le programme.

6. **Gestion des erreurs** : Si la commande n'est pas trouvée ou échoue, on affiche un message et on quitte proprement.
-

◆ 1. Découpage de la commande avec `split_command`

Lorsqu'on appelle `execute_command(cmd, envp)`, la première étape est de **séparer les arguments** de la commande.

Exemple

Commande d'entrée :

```
char *cmd = "ls -l /home";
```

Appel de `split_command(cmd)`, qui extrait :

```
cmd_args[0] = "ls"
cmd_args[1] = "-l"
cmd_args[2] = "/home"
```

Grâce à `split_command`, on obtient un tableau `cmd_args` contenant chaque argument de la commande.

◆ 2. Vérification d'un chemin absolu ou relatif

Après le découpage :

```
if (ft_strchr(cmd_args[0], '/'))
    handle_absolute_path(cmd_args, envp);
```

- Si `cmd_args[0]` contient `/` :
 - On suppose que l'utilisateur a donné un chemin absolu (`/bin/ls`) ou relatif (`./script.sh`).
 - `handle_absolute_path` :
 - Vérifie si le fichier existe (`access(cmd_args[0], F_OK)`)
 - Vérifie s'il est exécutable (`access(cmd_args[0], X_OK)`)
 - L'exécute avec `execve(cmd_args[0], cmd_args, envp)`

Si la commande **ne contient pas de /**, on doit chercher son chemin d'exécution.

♦ 3. Recherche du chemin dans le **PATH** (**find_command_path**)

Si la commande n'est pas un chemin absolu, on cherche où elle se trouve grâce à `find_command_path`.

Étapes de **find_command_path**

```
path_env = get_env_path(envp, "PATH=");
```

1. Récupère la variable **PATH** dans `envp` avec `get_env_path(envp, "PATH=")`.
2. Découpe **PATH** en plusieurs répertoires :

```
paths = ft_split(path_env, ':');
```

Exemple :

```
PATH = "/usr/local/bin:/usr/bin:/bin:/home/user/bin"
=> paths[0] = "/usr/local/bin"
    paths[1] = "/usr/bin"
    paths[2] = "/bin"
    paths[3] = "/home/user/bin"
```


3. Cherche la commande dans chaque dossier du **PATH** avec `search_in_paths` :

```
return (search_in_paths(paths, cmd));
```

- Construit chaque `full_path = paths[i] + "/" + cmd` avec `join_with_slash`.
- Vérifie si le fichier existe avec `access(full_path, F_OK)`.
- Renvoie le premier chemin valide trouvé.

Exemple

Pour la commande `"ls"`, la recherche se fait ainsi :

1. Teste `/usr/local/bin/ls` → Non trouvé
 2. Teste `/usr/bin/ls` → Trouvé 
 3. Retourne `"/usr/bin/ls"` comme chemin d'exécution.
-

◆ 4. Exécution de la commande

```
execve(path, cmd_args, envp);
```

- `execve` remplace le processus actuel par le programme situé à `path`.
 - Si `execve` échoue, on appelle `clean_exit` pour afficher un message et quitter.
-

◆ Résumé du fonctionnement

1. Découpe `cmd` en arguments (`split_command`).
 2. Si c'est un chemin absolu, on l'exécute directement.
 3. Sinon, on cherche la commande dans `$PATH` (`find_command_path`).
 4. On exécute la commande avec `execve`.
 5. Si une erreur se produit, on affiche un message et on quitte.
-

📌 Exemple final

Commande entrée :

```
execute_command("ls -l", envp);
```

Étapes exécutées :

Découpage :

```
cmd_args[0] = "ls"
```

```
cmd_args[1] = "-l"
```

- 1.
2. Pas de `/`, donc on cherche dans `PATH`.
3. Trouvé `/usr/bin/ls`.
4. Exécution avec `execve("/usr/bin/ls", cmd_args, envp);`.

✓ **Résultat** : Affiche la liste des fichiers avec `ls -l`.

Conclusion

Ce programme permet d'exécuter des commandes en les connectant via un pipe, imitant ainsi le fonctionnement du shell Unix. Il repose sur des concepts clés comme `fork()`, `execve()`, `pipe()`, `dup2()` et `waitpid()`.

1. Ouvre **infile** et **outfile**.
2. Crée un pipe.
3. Crée **cmd1** et redirige son **stdout** vers le pipe.
4. Crée **cmd2** et redirige son **stdin** depuis le pipe.
5. Exécute les commandes et redirige les sorties correctement.

Points Importants

- Descripteurs de Fichiers: Les descripteurs de fichiers sont des entiers qui identifient les fichiers ouverts dans un processus.
- Fermeture des Descripteurs: Il est important de fermer les descripteurs de fichiers inutilisés pour éviter les fuites de ressources.
- Synchronisation: Les pipes ne garantissent pas la synchronisation entre les processus. Des mécanismes supplémentaires comme les sémaphores peuvent être nécessaires pour une coordination plus complexe

Pour aller plus loin

1. **split_command()** (Découpage des arguments d'une commande)

 Code extrait :

```
char    **split_command(const char *str)
{
    char    **args;
    int     pos[3];
    int     count;

    initialize_positions_for_split_demesc(pos);
    count = count_args(str);
    args = ft_calloc(count + 1, sizeof(char *));
    if (!args)
        return (NULL);
```

```

while (str[pos[0]] && pos[1] < count)
{
    pos[0] = skip_whitespace(str, pos[0]);
    pos[2] = pos[0];
    pos[0] = skip_arg(str, pos[0]);
    if (pos[0] > pos[2] && !process_argument(args, str, pos))
    {
        ft_free_split(args);
        return (NULL);
    }
}
return (args);
}

```

Explication :

- Cette fonction découpe une **chaîne de commande** (par ex. "ls -l -a") en plusieurs morceaux ("ls", "-l", "-a").
- Elle compte d'abord le nombre d'arguments avec `count_args()`.
- Elle réserve de la mémoire (`ft_calloc`).
- Ensuite, elle **parcourt la chaîne de caractères** en ignorant les espaces (`skip_whitespace()`), puis en découpant les arguments (`skip_arg()`).

Lien avec d'autres fonctions :

- `initialize_positions_for_splitdemesc()` : Initialise les indices de parcours.
- `skip_whitespace()` : Ignore les espaces.
- `skip_arg()` : Traite chaque argument.
- `process_argument()` : Ajoute l'argument dans le tableau final.

Son rôle dans `pipex` :

Quand l'utilisateur passe "ls -l", cette fonction transforme "ls -l" en {"ls", "-l", NULL} avant que la commande ne soit exécutée avec `execve()`.

Cette fonction `split_command` est une version avancée d'un `split` classique en C, conçue pour gérer des commandes shell ou des chaînes d'arguments contenant des espaces, des guillemets et des échappements. Voyons en détail son fonctionnement et ses particularités :

♦ Objectif de la fonction `split_command`

L'objectif principal est de découper une chaîne `str` en arguments séparés par des espaces, tout en respectant les guillemets (simples `'` ou doubles `"`). Contrairement à `strtok()` ou à une simple fonction de découpage par espace, celle-ci :

1. **Ignore les espaces superflus** au début et entre les arguments.
2. **Gère les guillemets** : si un argument est entouré de guillemets ("`argument avec espace`" ou '`autre exemple`'), il est considéré comme un seul élément.
3. **Nettoie les guillemets** : après la séparation, les guillemets sont retirés.
4. **Alloue dynamiquement un tableau de chaînes** (`char **`) contenant les arguments.

♦ Déroulement de `split_command`

1. Initialisation :

- On initialise un tableau `args` pour stocker les arguments.
- On compte d'abord combien d'arguments sont présents avec `count_args()`.
- On réserve de la mémoire pour `args` avec `ft_calloc()`.

2. Parcours de la chaîne et découpage des arguments :

- On commence à la position `pos[0]` et on saute les espaces (`skip_whitespace()`).
- `pos[2]` mémorise le début du nouvel argument.
- On avance jusqu'à la fin de l'argument avec `skip_arg()`.
- Si un argument est trouvé, il est extrait et stocké grâce à `process_argument()`.

3. Ajout de l'argument :

- `add_arg()` extrait l'argument avec `ft_substr()`.
- `clean_quotes()` supprime les guillemets autour de l'argument si nécessaire.
- L'argument est stocké dans `args`.

♦ Détails des Fonctions Clés

❶ `count_args(const char *str)`

- Compte combien d'arguments il y a dans `str`.
- Ignore les espaces et gère les guillemets.

❷ `skip_whitespace(const char *str, int pos)`

- Saute les espaces et les tabulations.

❸ `skip_arg(const char *str, int pos)`

- Avance jusqu'à la fin d'un argument.
- Prend en compte les guillemets via `handle_quote()`.

❹ `handle_quote(const char *str, int i, char quote)`

- Si un guillemet (' ou ") est trouvé, il avance jusqu'à la fermeture du guillemet.

❺ `clean_quotes(char *str)`

- Supprime les guillemets des arguments.

❻ `process_argument(char **args, const char *str, int pos[3])`

- Copie l'argument trouvé dans `args`.
- Nettoie les guillemets avec `clean_quotes()`.

♦ Différences avec un `split` classique (`strtok()` ou `strsep()`)

1. ☒ **Gère les espaces multiples** : Ne produit pas d'arguments vides si plusieurs espaces sont présents.
2. ☒ **Prend en charge les guillemets** : Un argument entre guillemets reste un seul élément.
3. ☒ **Alloue dynamiquement la mémoire** pour stocker les arguments.
4. ☒ **Supprime les guillemets inutiles** : L'utilisateur ne récupère pas d'arguments entourés de " ou '.

Cette fonction `split_command` est une version avancée de `split()`, adaptée pour analyser des commandes shell. Elle gère les guillemets, ignore les espaces en trop et assure une bonne allocation mémoire. C'est très utile pour un interpréteur de commandes ou un mini-shell en C

2. `find_command_path()` (Trouver le chemin d'un programme)

 Code extrait :

```
char    *find_command_path(char *cmd, char **envp)

{
    char    *path_env;

    char    **paths;

    path_env = get_env_path(envp, "PATH=");

    if (!path_env)

        return (get_valid_path(cmd, envp));

    paths = ft_split(path_env, ':');

    if (!paths)

        return (NULL);

    return (search_in_paths(paths, cmd));
}
```

 Explication :

1. Recherche la **variable d'environnement PATH** (qui contient tous les dossiers où chercher des programmes).
2. Découpe ce **PATH** en une **liste de dossiers** (`ft_split(path_env, ':')`).
3. Recherche le programme dans ces **dossiers un par un** (`search_in_paths()`).

 Lien avec d'autres fonctions :

- `get_env_path()` : Récupère la variable `PATH` depuis l'environnement.
- `ft_split()` : Découpe la variable `PATH` en une liste de dossiers.
- `search_in_paths()` : Teste chaque dossier pour voir si le programme y existe.

✓ **Son rôle dans `pipex` :**

Si tu tapes "`ls`", le programme va chercher `/bin/ls` en testant chaque dossier du `PATH` jusqu'à trouver `ls`.

3. `search_in_paths()` (Tester les chemins pour trouver la commande)

 Code extrait :

```
char    *search_in_paths(char **paths, char *cmd)

{

    char    *full_path;

    int      i;

    i = -1;

    while (paths[++i])

    {

        full_path = join_with_slash(paths[i], cmd);

        if (!full_path)

            return (ft_free_split(paths), NULL);

        if (access(full_path, F_OK) == 0)
```

```

        return (ft_free_split(paths), full_path);

    free(full_path);

}

return (ft_free_split(paths), NULL);

}

```

Explication :

- Cette fonction prend la **liste des chemins** et **teste** si le programme `cmd` est présent dans un de ces chemins (`/usr/bin`, `/bin`, etc.).
- Elle teste avec `access(full_path, F_OK)`, qui vérifie si le fichier existe.

Lien avec d'autres fonctions :

- `join_with_slash()` : Ajoute un `/` entre le chemin (`/bin`) et la commande (`ls`).
- `access()` : Vérifie l'existence du fichier.

Son rôle dans `pipex` :

Si tu tapes "`ls`", `search_in_paths()` va tester :

- `/bin/ls`  (trouvé ici)
- `/usr/bin/ls`  (non utilisé)

4. `handle_absolute_path()` (Gérer les chemins absolus)

```

void    handle_absolute_path(char **cmd_args, char **envp)

{
    if (access(cmd_args[0], F_OK) == -1)

```

```

        clean_exit(cmd_args, cmd_args[0], "No such file or directory",
127);

    if (access(cmd_args[0], X_OK) == -1)

        clean_exit(cmd_args, cmd_args[0], "Permission denied", 126);

    execve(cmd_args[0], cmd_args, envp);

    clean_exit(cmd_args, cmd_args[0], "execve failed", 1);

}

```

Explication :

- Vérifie si la commande entrée est un **chemin absolu** (`/usr/bin/ls`).
- Vérifie que le fichier existe (`F_OK`).
- Vérifie qu'il est **exécutable** (`X_OK`).
- Exécute directement la commande avec `execve()`.

Son rôle dans `pipex` :

Si tu lances `./mon_programme`, cette fonction s'assure que le fichier existe et est exécutable avant de le lancer.

5. `get_env_path()` (Trouver une variable d'environnement)

```

char    *get_env_path(char **envlp, const char *name)

{
    int i;

    i = 0;

    while (envlp[i])

    {

```



```

    if (ft_strncmp(envlp[i], name, ft_strlen(name)) == 0)

        return (envlp[i] + ft_strlen(name));

    i++;

}

return (NULL);

}

```

Explication :

- Parcourt toutes les **variables d'environnement**.
- Retourne la valeur de **PATH** (/bin:/usr/bin).

Son rôle dans **pipex** :

C'est cette fonction qui **trouve le PATH** pour que **find_command_path()** puisse chercher **ls**.

Schéma d'un pipe :

[Processus 1] → (pipefd[1]) -----> (pipefd[0]) → [Processus 2]

- **pipefd[1]** → **Écriture** dans le pipe
- **pipefd[0]** → **Lecture** depuis le pipe
-

Conclusion : Comment tout s'imbrique ?

➡ Sépare "ls -l" en { "ls", "-l", NULL }

➡ Cherche "ls" dans /bin/ls, /usr/bin/ls, etc.

➡ Vérifie que "ls" existe et est exécutable.

➡ Lance "ls -l" avec les bons arguments.

cmd1 | cmd2 ! 🚀

en C

1 Qu'est-ce que Pipex ?

Pipex est un programme en C qui reproduit le comportement suivant d'un shell :

```
< infile cmd1 | cmd2 > outfile
```

- Il lit un fichier **infile**.
- Il exécute une première commande **cmd1**.
- Il transmet la sortie de **cmd1** en entrée de **cmd2** grâce à un pipe.
- Il écrit le résultat final dans un fichier **outfile**.

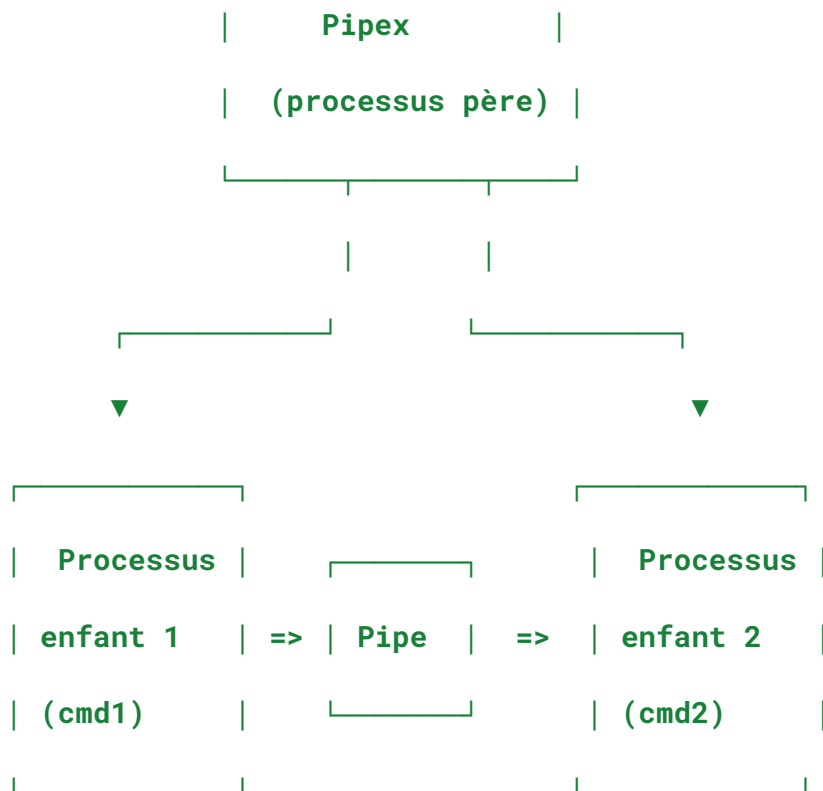
2 Vue globale des processus et des flux de données

👉 Le programme crée deux processus enfants pour exécuter `cmd1` et `cmd2`, et il utilise un `pipe()` pour la communication entre eux.

 **Schéma global du fonctionnement de Pipex :**

plaintext





- **cmd1** écrit dans le pipe.
- **cmd2** lit depuis le pipe.
- Le processus parent attend la fin des deux commandes (**waitpid()**).

3 Étapes détaillées du programme

Voici comment Pipex fonctionne en interne avec **fork()**, **pipe()**, **dup2()** et **execve()**.

● Étape 1 : Initialisation

- Vérifie si le nombre d'arguments est correct.
- Ouvre **infile** en lecture et **outfile** en écriture.
- Crée un pipe avec **pipe(pipex->pipefd)**.

👉 À ce stade, le pipe est créé mais vide.

● Étape 2 : Création du premier processus enfant (cmd1)

- Le parent `fork()` un processus enfant.
- L'enfant redirige :
 - `stdin` (entrée standard) vers `infile` avec `dup2(exec->infile_fd, STDIN_FILENO)`.
 - `stdout` (sortie standard) vers l'entrée du pipe avec `dup2(pipex->pipefd[1], STDOUT_FILENO)`.
- Ferme les fichiers inutiles et exécute `cmd1` avec `execve()`.

👉 `cmd1` lit dans `infile` et écrit dans le pipe.

🟡 Étape 3 : Création du deuxième processus enfant (`cmd2`)

- Le parent `fork()` un second processus enfant.
- L'enfant redirige :
 - `stdin` vers la sortie du pipe avec `dup2(pipex->pipefd[0], STDIN_FILENO)`.
 - `stdout` vers `outfile` avec `dup2(exec->outfile_fd, STDOUT_FILENO)`.
- Ferme les fichiers inutiles et exécute `cmd2` avec `execve()`.

👉 `cmd2` lit depuis le pipe et écrit dans `outfile`.

🔴 Étape 4 : Fermeture et synchronisation

- Le processus parent ferme les descripteurs de pipe devenus inutiles.
- Il attend la fin des deux commandes avec `waitpid()`.

👉 Tout est exécuté, et Pipex se termine ! 🎉

Qu'est-ce qu'un processus zombie ?

Un **processus zombie** est un processus enfant qui a terminé son exécution, **mais dont le parent n'a pas récupéré son statut de sortie avec `wait()` ou `waitpid()`.**

⚠ Pourquoi est-ce un problème ?

- Même si le processus enfant est terminé, son **PID reste occupé** dans la table des processus.
- Si trop de processus zombies s'accumulent, cela peut épuiser les PID disponibles et causer des problèmes au système.

1. Parent et Enfants dans un Programme C avec `fork()`

Lorsque tu appelles la fonction `fork()` dans un programme C :

1. **Le processus parent :**
 - C'est le processus original qui appelle `fork()`.
 - Après l'appel à `fork()`, il continue son exécution.
2. **Le processus enfant :**
 - C'est une **copie exacte du parent** créée par `fork()`.
 - Il commence à exécuter le code **juste après l'appel à `fork()`**.
3. **Comment les différencier ?**
 - `fork()` retourne **0** dans le **processus enfant**.
 - `fork()` retourne un **PID (Process ID)** positif dans le **processus parent** (le PID du processus enfant).
 - Si `fork()` retourne **-1**, cela signifie qu'une **erreur** est survenue.
 -

2. Bonnes pratiques avec les pipes

- **Toujours fermer les extrémités inutilisées :**
 - Le processus qui **lit** doit fermer `pipefd[1]`.
 - Le processus qui **écrit** doit fermer `pipefd[0]`.
- **Éviter les blocages :**
 - Si un processus lit depuis un pipe vide, il se bloque jusqu'à ce qu'il y ait des données.
 - Si un processus écrit dans un pipe sans lecteur, il déclenche une erreur **2**.

3. Comprendre les processus

✓🔑 `fork()`

Prototype :

```
pid_t fork(void);
```

- **Retourne :**
 - **0** → Dans le processus enfant.
 - **> 0** → PID du processus enfant dans le processus parent.
 - **-1** → Erreur.

🔄 **Fonctionnement :**

- `fork` crée un **nouveau processus enfant** qui est une copie exacte du processus parent.
- Les deux processus continuent leur exécution **à partir de la ligne suivant le `fork()`**.

✓🔑 `execve()`

Prototype :

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

- **filename** : Chemin absolu/relatif du programme à exécuter (`/bin/ls`).
- **argv** : Tableau des arguments passés au programme (`{"ls", "-l", NULL}`).
- **envp** : Tableau des variables d'environnement.

📝 **Fonctionnement :**

- `execve` remplace le code et l'espace mémoire du processus appelant par le programme spécifié.
- **Ne retourne jamais en cas de succès.**
- En cas d'échec, retourne `-1`.

✓🔑 `wait()`

Prototype :

```
pid_t wait(int *status);
```

- **Attend qu'un processus enfant se termine.**
- Retourne le **PID de l'enfant terminé**.
- Le paramètre `status` permet de récupérer le **code de sortie** du processus enfant.

Exemple :

```
int status;

pid_t child_pid = wait(&status);

if (WIFEXITED(status)) {

    printf("Enfant terminé avec le code : %d\n", WEXITSTATUS(status));

}
```

fork : Crée deux processus enfants.

execve : Remplace l'espace mémoire par le programme (`cmd1`, `cmd2`).

waitpid : Empêche les **processus zombies** en attendant les enfants.

dup2 : Redirige les descripteurs de fichiers (`STDIN`, `STDOUT`).

Fermeture des descripteurs inutilisés : Indispensable pour éviter les fuites.

✅ 1. Qu'est-ce que **dup2** ? *Duper le dup2*

Prototype :

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



Fonctionnement :

- **oldfd** : Le **descripteur de fichier existant** que tu veux dupliquer/rediriger.
- **newfd** : Le **descripteur de fichier cible** où tu veux rediriger **oldfd**.
- **Retourne :**
 - Un entier positif en cas de **succès**.
 - **-1** en cas d'**échec** (et définit **errno**).

✅ 2. À quoi ça sert ?

dup2 est utilisé pour **rediriger des descripteurs de fichiers standard** :

- **STDIN_FILENO (0)** → Entrée standard (par défaut le clavier).
- **STDOUT_FILENO (1)** → Sortie standard (par défaut le terminal).
- **STDERR_FILENO (2)** → Sortie des erreurs (par défaut le terminal).



Exemple de redirection :

- Lire à partir d'un fichier au lieu du clavier (**STDIN**).
- Écrire dans un fichier au lieu du terminal (**STDOUT**).

✓ 3. Exemple Basique avec **dup2**



Rediriger **STDOUT** vers un fichier

```
#include <stdio.h>

#include <fcntl.h>

#include <unistd.h>

int main() {

    int file_fd = open("output.txt", O_WRONLY | O_CREAT | O_TRUNC,
0644);

    if (file_fd == -1) {

        perror("Erreur ouverture fichier");

        return 1;

    }

    // Rediriger STDOUT (1) vers le fichier

    if (dup2(file_fd, STDOUT_FILENO) == -1) {

        perror("Erreur dup2");

        close(file_fd);

    }
```



```

        return 1;

    }

    // Tout ce qui est écrit sur STDOUT ira dans "output.txt"

    printf("Ce message va dans le fichier, pas sur le terminal.\n");

    close(file_fd); // Fermer le fichier après redirection

    return 0;
}

```

✓ Explications :

1. **Ouverture du fichier :**
 - Le fichier `output.txt` est ouvert en écriture.
 2. **Redirection avec `dup2` :**
 - `dup2(file_fd, STDOUT_FILENO)` remplace le descripteur **STDOUT (1)** par le **descripteur du fichier**.
 3. **Écriture avec `printf` :**
 - Maintenant, tout ce qui est écrit avec `printf` ira dans `output.txt`.
-

✓ 4. Pourquoi `dup2` est important dans Pipex ?

Dans Pipex, tu as besoin de rediriger :

- **L'entrée standard (STDIN)** vers `infile` (fichier source).
- **La sortie standard (STDOUT)** vers un `pipe` ou un `outfile` (fichier destination).

🔧 Exemple simplifié :

Schéma :

```

[ infile ] → cmd1 → (pipefd[1]) -----> (pipefd[0]) → cmd2 → [
outfile ]

```

Avec `dup2` :

Pour **cmd1** :

```
dup2(infile, STDIN_FILENO); // STDIN vient de infile

dup2(pipefd[1], STDOUT_FILENO); // STDOUT va vers le pipe
```

Pour **cmd2** :

```
dup2(pipefd[0], STDIN_FILENO); // STDIN vient du pipe

dup2(outfile, STDOUT_FILENO); // STDOUT va vers outfile
```

✓ 5. Schéma Visuel de **dup2**

Avant redirection :

```
STDIN (0) → Clavier

STDOUT (1) → Terminal
```

Après `dup2(infile, STDIN_FILENO)` :

```
STDIN (0) → infile
```

Après `dup2(pipefd[1], STDOUT_FILENO)` :

```
STDOUT (1) → pipefd[1]
```

Maintenant, quand **cmd1** lit **STDIN**, elle lit depuis **infile**, et quand elle écrit sur **STDOUT**, elle écrit dans le **pipe**.

✓ 6. Que se passe-t-il exactement avec **dup2** ?

1. Ferme **newfd** s'il est déjà ouvert.
2. Fait pointer **newfd** vers le même fichier que **oldfd**.
3. Les deux descripteurs partagent maintenant le même fichier.

Exemple technique :

```
int fd = open("fichier.txt", O_RDONLY);

dup2(fd, STDIN_FILENO); // STDIN (0) est maintenant lié à fichier.txt

close(fd); // On ferme fd car STDIN utilise déjà le même fichier
```

✓ 7. Bonnes pratiques avec **dup2** :

Toujours vérifier le retour de **dup2**.

```
if (dup2(fd, STDIN_FILENO) == -1) {  
  
    perror("Erreur dup2");  
  
    exit(EXIT_FAILURE);  
}
```

Fermer les descripteurs inutilisés après redirection.

```
close(fd);
```

Ne pas oublier les erreurs de redirection :

Une erreur ici signifie que ton programme ne lira/écrira pas correctement.

✓ 8. Récapitulatif rapide :

Récapitulatif des cas d'utilisation de **-1** :

Fonction	Signification de -1
open	Erreur à l'ouverture du fichier (ex: permission, fichier inexistant).
read	Erreur pendant la lecture (ex: problème matériel, descripteur invalide).
write	Erreur pendant l'écriture (ex: espace disque insuffisant).
close	Erreur pendant la fermeture du fichier (ex: descripteur invalide).

Fonction	Description
dup2	Redirige un descripteur de fichier existant (oldfd) vers un autre (newfd).
STDIN	Entrée standard (0) : souvent redirigée depuis un fichier ou un pipe.
STDOUT	Sortie standard (1) : souvent redirigée vers un fichier ou un pipe.
Exemple	<code>dup2(infile, STDIN_FILENO)</code> : lit depuis le fichier.

1. Pourquoi gérer les erreurs et les descripteurs avec soin ?

🔑 Gestion des erreurs : Pourquoi ?

- Une erreur non gérée peut provoquer des comportements imprévisibles du programme.
- Les erreurs courantes incluent :
 - L'échec d'une ouverture de fichier (**open**).
 - Une redirection incorrecte avec **dup2**.
 - Un problème avec la création du **pipe** ou des processus enfants (**fork**).

🔑 Descripteurs de fichiers : Pourquoi les fermer ?

- Chaque **descripteur de fichier ouvert utilise une ressource système limitée**.
 - Si tu ne fermes pas les descripteurs inutilisés, cela peut entraîner :
 - **Des fuites de ressources.**
 - **Un comportement inattendu du programme.**
-

✓ 1. Pourquoi est-il important de mettre les pointeurs à **NULL** après **free** ?

📖🔑 Explication :

Lorsque tu utilises la fonction **free()** pour libérer un pointeur, la mémoire qu'il pointait est **rendue au système**, mais le pointeur lui-même **contient toujours l'adresse mémoire** qu'il pointait auparavant. Cela le transforme en **pointeur "dangling" (pendant)**.

- **Pointeur dangling** : C'est un pointeur qui ne pointe plus vers une mémoire valide.
- **Problèmes possibles** :
 - **Double Free** : Si tu appelles **free()** une deuxième fois sur ce pointeur, cela provoque un comportement indéfini et peut corrompre la mémoire.
 - **Segmentation Fault** : Accéder à un pointeur dangling peut causer un plantage.
 - **Fuites mémoire cachées** : Le programme peut croire que la mémoire est encore valide.

✓🛡️ Bonnes Pratiques :

Après un **free**, toujours **assigner le pointeur à NULL** :

```
free(ptr);  
ptr = NULL;
```

🔄 Pourquoi ?

1. **NULL** est une adresse spéciale qui indique que le pointeur ne pointe vers rien.
2. Les appels ultérieurs à **free(NULL)** sont sûrs et n'ont aucun effet.

Exemple d'une mauvaise pratique :

```
char *str = malloc(10);  
free(str);  
free(str); // ⚠️ Erreur : Double free
```

Exemple sécurisé :

```
char *str = malloc(10);  
free(str);  
str = NULL; // ✓ Sécurisé  
free(str); // Sans effet, car str est NULL
```

1 Commande Shell équivalente

Dans un shell Unix classique, une commande avec un pipe ressemble à ceci :

```
< infile cat | grep "texte" > outfile
```

Ce qui se passe :

- `< infile` : On lit les données de `infile`.
- `cat` : Affiche le contenu du fichier.
- `| grep "texte"` : Filtre les lignes contenant `"texte"`.
- `> outfile` : Écrit le résultat dans `outfile`.

2 Test avec `pipex`

Si ton programme `pipex` fonctionne comme attendu, tu devrais pouvoir exécuter la commande suivante :

```
./pipex infile "cat" "grep texte" outfile
```

Vérifie ensuite le contenu de `outfile` : `cat outfile`

Il devrait contenir uniquement les lignes de `infile` qui contiennent `"texte"`.

Test avec `echo`

Si tu n'as pas de fichier d'entrée, tu peux tester directement avec `echo` en redirigeant vers un fichier :

```
echo "Hello\nWorld\nPipex" > testfile
./pipex testfile "cat" "tr a-z A-Z" outfile
cat outfile
```

Ici, `tr a-z A-Z` convertit le texte en majuscules.

Si tu veux tester des erreurs (ex: fichier inexistant, commande invalide), essaie :

```
./pipex nonexistent "cat" "wc -l" outfile
```

renvoie un message d'erreur.