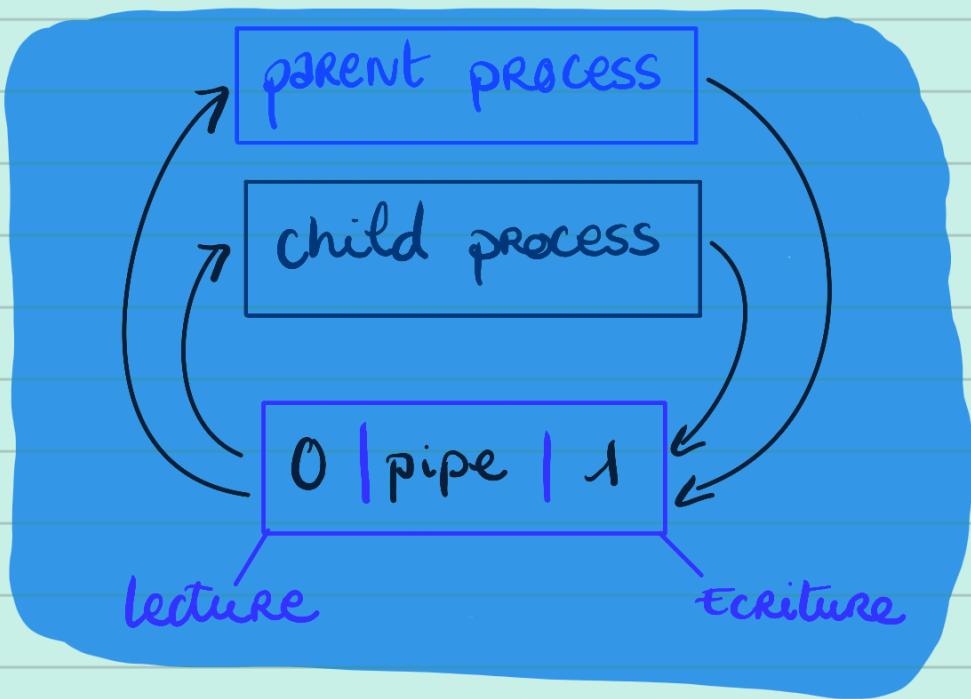


PIPEX

→ Un pipe = tube (fr) est une section de mémoire partagée qui facilite la communication entre processus.

Ce canal est unidirectionnel = un extrémité de lecture et extrémité d'écriture.

Un processus peut écrire sur l'extr. d'écriture ; ces données seront mises en mémoire tampon, jusqu'à ce qu'elles soient lues par un autre processus depuis l'extrém. de lecture du tube.



Un pipe prend la forme d'un fichier qui réside en dehors du système de fichiers et n'a pas de nom d'attributs particulier.

Pourtant on le manipule comme un fichier grâce à ses deux descripteurs de fichier.

est un nombre entier positif, l'index d'un fichier dans une structure de données contenant les informations de tous les fichiers ouverts dans le système.

⇒ Quand on crée un pipe, on reçoit deux descripteurs

de fichier vers le même tube(pipe), l'un est ouvert en mode lecture seule et l'autre écriture seule.

⚠ Il existe une limite de taille pour un pipe (varie selon OS)

→ bibliothèque <unistd.h>

↳ prototype int pipe(int pipefd[2]);

→ tableau de deux entiers où stocker les deux descripteurs de fichiers.

→ représente les deux bouts du tube



- pipefd [0] = lecture
- pipefd [1] = écriture

→ L'appel pipe ouvrira les fd du pipe et les renseignera dans ce tableau qu'on lui fournira en paramètre.

pipe → return 0 si OK

↳ return -1 si ERROR (voir errno)

⇒ Pour établir une communication inter-processus entre un processus parent et son processus enfant il nous faudra donc créer un pipe.

Quand on crée ensuite l'enfant, il aura un duplicata des fd de ce pipe, puisqu'un processus enfant est le clone du parent.

Ainsi, l'enfant sera en mesure de lire depuis pipefd[0] les informations fournies par son parent dans pipefd[1] et inversement.

lire et écrire dans un pipe =

↳ Afin de mettre des données / ou de les récupérer dans le descripteur de fichier du pipe, on se sert de `read` et `write`.



→ Si un processus tente de lire depuis un pipe vide, `read` bloquera le processus jusqu'à ce que les données soient écrites dans le pipe.

→ À l'inverse si le pipe est plein (limite de sa capacité), `write` bloquera le processus jusqu'à ce qu'assez de données soient lues pour y écrire alors.

Fermer un pipe =

↳ les borts de lecture et d'écriture d'un pipe se ferment avec l'appel système `close` -



→ lorsque tous les descripteurs de fichier qui font référence au bort écriture du pipe sont fermés, un process qui tentera le bort de lecture verra le caractère EOF et `read` renvoie 0.



→ À l'inverse, si tous les descripteurs de fichier pr le bort de lecture sont fermés et qu'un process tente d'y écrire, `write` lancera le signal SIGPIPE on echouera avec EPIPE dans errno.

⇒ Pour s'assurer que les processus reçoivent bien les indications de terminaison (EOF, SIGPIPE/EPIPE) il est primordial de fermer tous les descripteurs inutilisés.

↳ ./pipex file1 cmd1 cmd2 file2
↳ \$> <file1 cmd1 | cmd2 > file2

exemple : ./pipex infile "ls -l" "wc -l" outfile

→ Ce projet consiste à gérer des pipes

→ fonctions autorisées : open / close / read / write /
malloc / free / exit

perror (affiche un message descriptif à "stderr")

strerror (renvoie un pointeur sur une chaîne qui explique l'erreur)

dup (crée une copie du file descriptor oldfd)

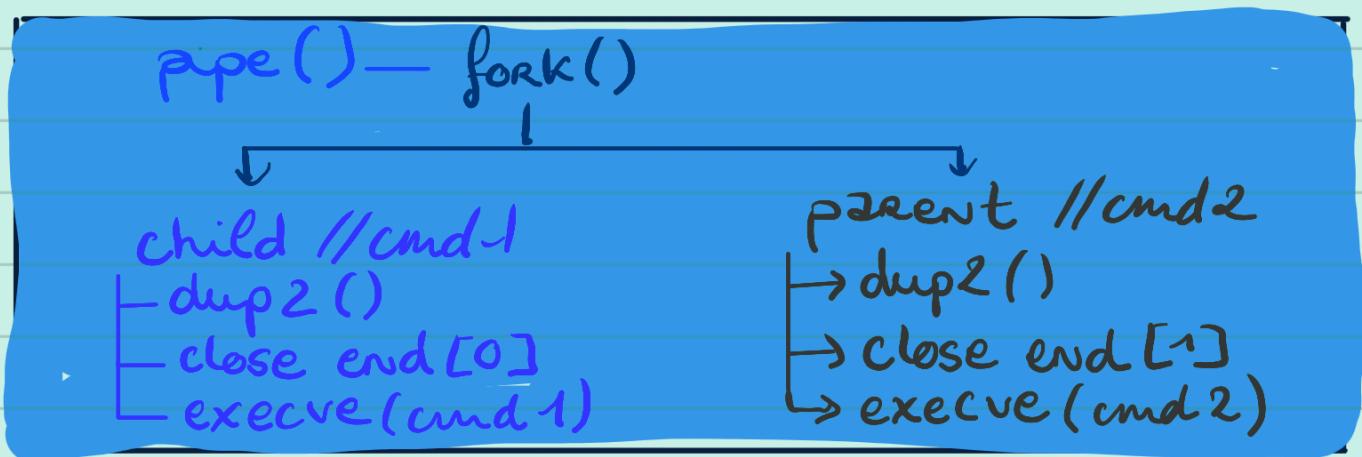
dup2 (crée un nouveau fd qui est la copie du vieux fd et ferme forcément le nouveau avant)

fork (Crée un nouveau processus en duplant celui déjà existant)

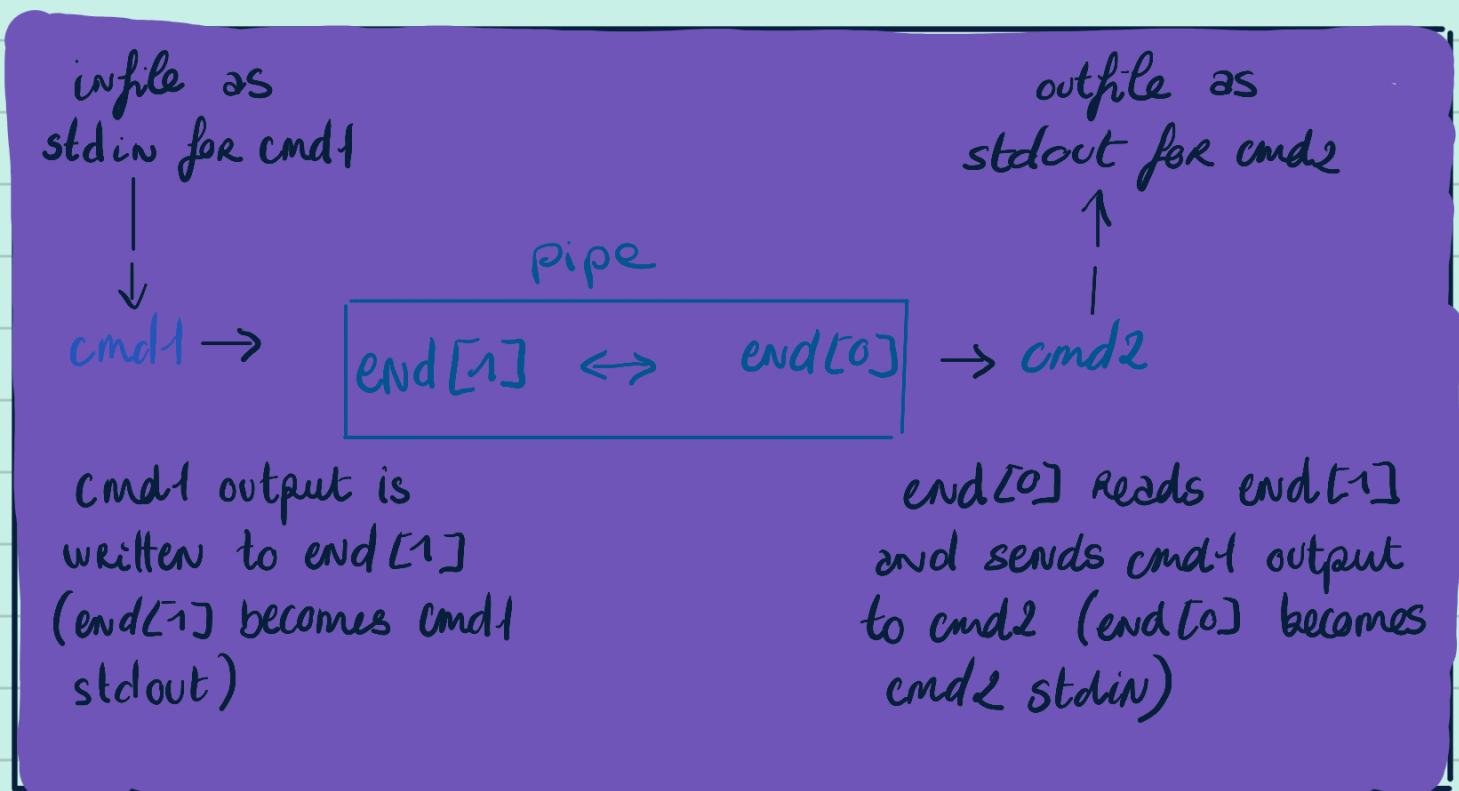
pipe (Crée un pipe, une data indirectionnelle pour les communications inter-processus)

- GENERAL IDEA -

- We read from infile
- Execute cmd1, with infile as input
- Send the output to cmd2, which will write to outfile



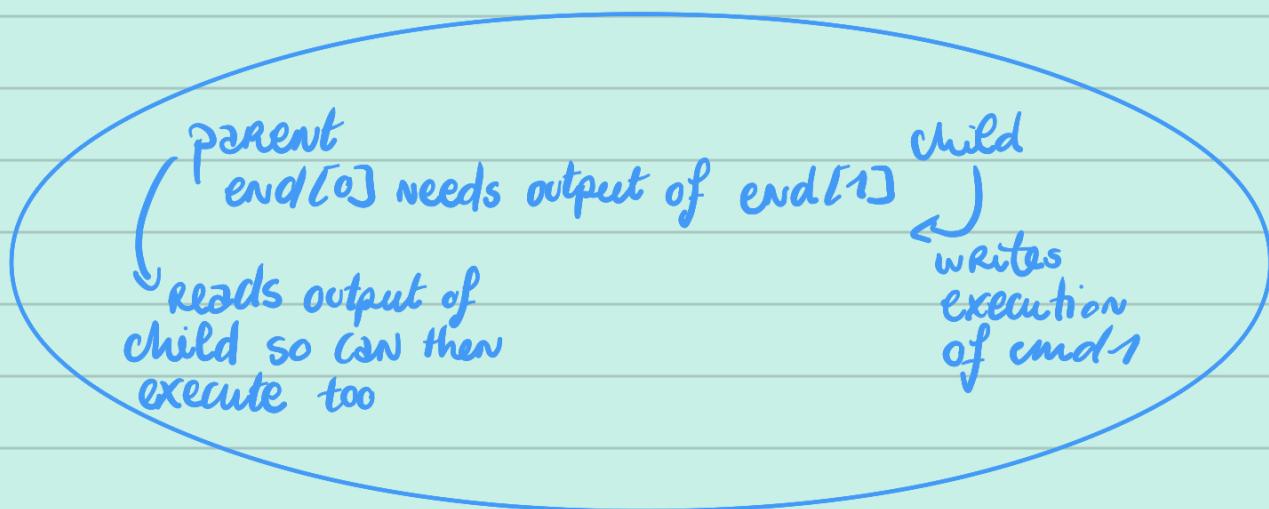
pipe() sends the output of the first execve(cmd1) as input to the second execve(cmd2);
fork() runs two processes (= commands) in one single process;
dup2() swaps our files with stdin and stdout.



`pipe()` → takes an array of two → int `end[2]`
→ `end[0]` is visible in `end[1]`, can communicate
→ assigns an fd to each end.
↳ `end[1]` writes to its own fd, same for `end[0]`

`fork()` → splits process in two processes*
→ returns 0 for the child process
→ returns !=0 for the parent process
→ Returns -1 in case of error.
* two parallel, simultaneous processes

{ `end[1]` is the child
`end[0]` is the parent
↳ the child writes, while the parent reads
(for something to be read, it must be written first)
↳ so `cmd1` will be executed by the child
and `cmd2` by the parent.



• On linux we can check fds currently open with command
`ls -la /proc/$$/fd (0,1,2 are by default assigned to
stdin, stdout and stderr).`

| | |
|---|------------------|
| 0 | stdin |
| 1 | stdout |
| 2 | stderr |
| 3 | infile //open() |
| 4 | outfile //open() |
| 5 | end[0] |
| 6 | end[1] |

`cmd1 = child → stdin = infile = input`
↳ `stdout = end[1] = output`

`cmd2 = parent → stdin = end[0] = reads end[1] output`
↳ `stdout = outfile = write output`

`dup2()` → swap fds to stdin/stdout
→ `int dup2(int fd1, int fd2);`
↳ close fd2 and duplicate the value of fd2
to fd1 - or redirect fd1 to fd2.

CHILD_process(f1, cmd1) in pseudo-code

→ // protection if dup2 failed (< 0).
// dup2 close stdin, f1 becomes the new stdin
dup2(f1, STDIN_FILENO); // need f1 to be execve() input
dup2(end[1], STDOUT_FILENO); // need end[1] to be
execve() stdout
close(end[0]); // always close the end of the pipe you
don't use, as long as the pipe

⚠ Protect every function's return like read,
write, pipe etc = handle every situation

`fork()` → after the line where we call `fork()`, everything is going to be executed twice.
→ one will be executed in main process and the second in the child process.
↳ the id in the child process will always be 0

→ clone the calling process, creating an exact copy. Return -1 for errors, 0 to the new process, and the process ID of the new process to the old process.

→ So we have to protect the function's return
= if (`id == 0`) → do instructions with child process
= else → do instructions with main process
↳ OR ERROR ?

⚠ → every time we print something on the screen/standard output, because it has an internal buffer, if we print a lot of things very quickly it's not gonna automatically print them on the cause but going to wait to get all the input and then print the whole buffer...

that is an example of why do we need the `fflush` function

→ `fflush(stdout)`

 → the order of the split process output after the fork() is chaotic and isn't guaranteed

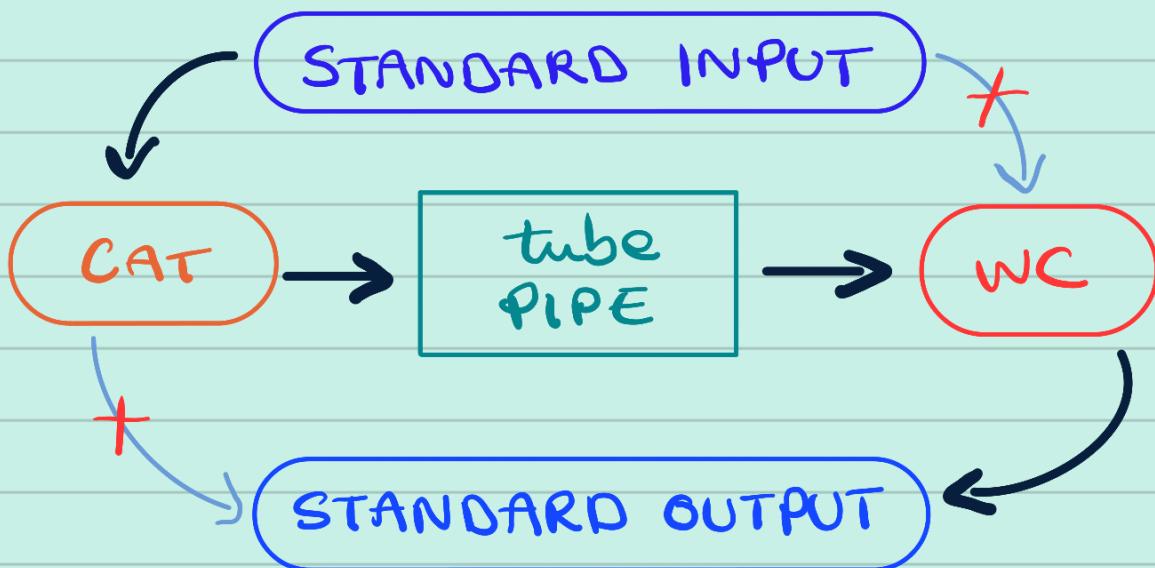
↓ that's why we need wait() function

↳ example : "stop execution with the main process until the child process is running execution."

```
if (id != 0) //if we are in the main process  
    wait(); //
```

→ Que fait l'opérateur " | " pipe dans la ligne de commande " cat text.txt | wc -l " ?

Il crée un pipe et deux processus enfant, un pour la commande 'cat' et un pour 'wc'. Ensuite il fait une redirection de la sortie standard de 'cat' vers l'entrée standard de 'wc'.



→ On pourrait reproduire ce comportement en dupliquant le bout d'écriture du pipe sur la sortie standard du premier enfant, et le bout de lecture sur l'entrée standard du second enfant.

= dup2

dup2() → dupliquer des FD.

→ parfois il est utile de dupliquer un FD pour en faire la sauvegarde ou le remplacer.

C'est le cas dans le contexte de redirection de l'entrée/sortie standard vers un fichier.

(int dup2(int oldfd, int newfd);

→ Prend en paramètre le descripteur de fichier qu'on veut dupliquer = oldfd et renvoie le nouveau descripteur de fichier / ou -1 si erreur.
le nouveau descripteur de fichier = newfd.

→ Si newfd est utilisé avant d'être transformé en une copie de oldfd, la fonction va tenter de fermer newfd.

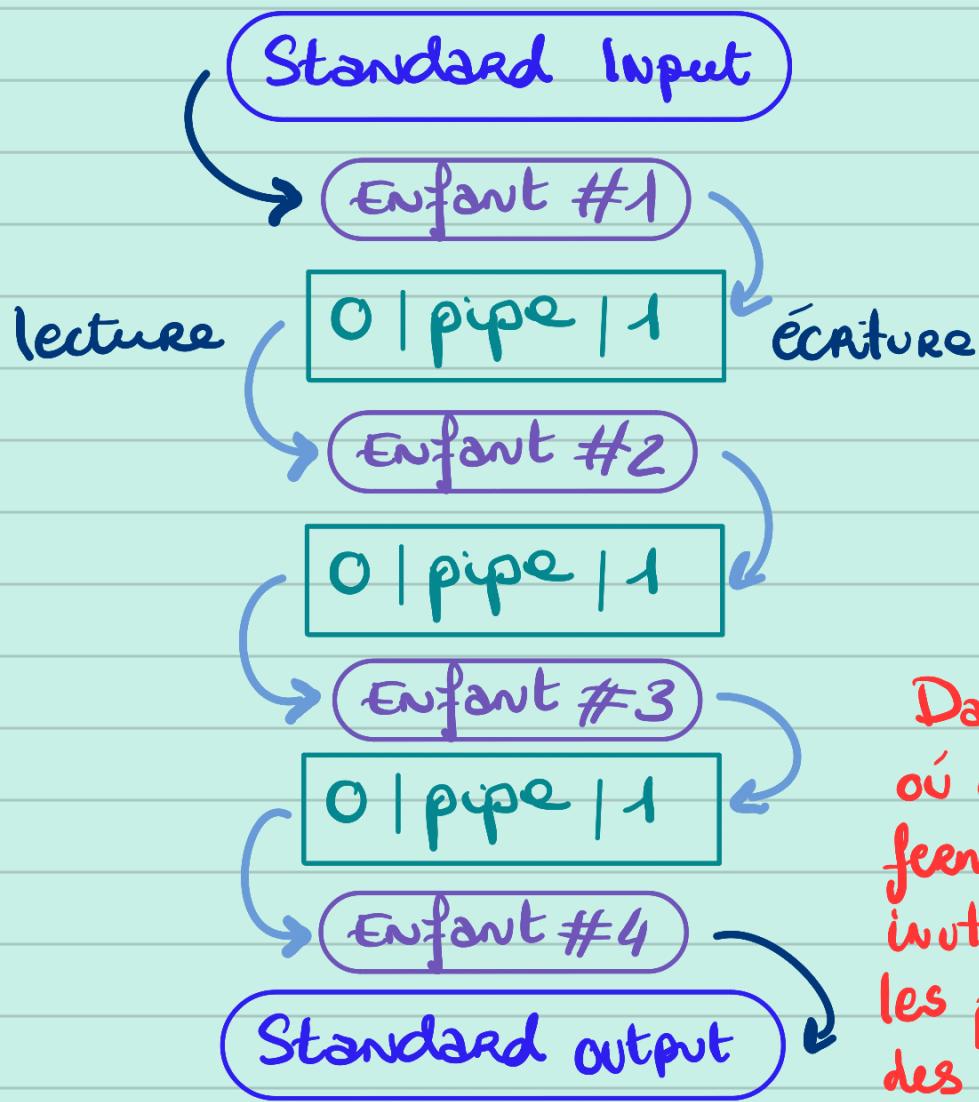
→ Si newfd et oldfd sont identiques et valides, dup2 renverra newfd sans rien faire d'autre.

→ pipeline = série de pipes.

→ Si on se contente d'utiliser un seul pipe pour l'entrée/sortie de tous les processus enfants = PROBLÈMES → Car les process sont exécutés simultanément.

Donc soit utiliser wait() ou bien créer un pipe pour chaque processus enfant, moins 1.

Le premier peut écrire dans la sortie de son propre tube, le deuxième lire depuis le tube du premier et écrire dans son propre tube ensuite et ainsi de suite.



Dans la mesure où on oublie pas de fermer tous les fils utilisés de tous les pipes ds chacun des processus enfant

1 - Creating a pipe

- tab of len 2 = fd
- call pipe(fd)
- ↳ protect if it returns -1 = exit

2 - Creating two processes

- var of type int = id-proc1
- call fork() for id-proc1
- ↳ protect if it returns less than zero = exit

- 1 → If id-proc1 is equal to zero = child1 → cmd1
 - ↳ Need redirection of the cmd1's output to be the input of cmd2
= dup2(fd[1], STDOUT_FILENO)
where ↑ → ↑
wrote output
 - ↳ Need to close fd[0] because isn't used
= close(fd[0])
And we did duplicated fd[1] with dup2 so we need to close the original = close(fd[1])
 - ↳ Finally we can execute cmd1 with execve()

- New var of type int = id-proc2
- call fork() for ↑
- ↳ protect if returns less than zero

2 → If id-proc2 is zero = child 2 → cmd2

↳ Need redirection of the cmd2
= dup2(fd[0], STDIN_FILENO)

↳ close(fd[0]) and close(fd[1])

↳ Then execute cmd2

→ Close both fd[2] because the main process is still using it, and the program won't stop if we forget it.

→ Use the function waitpid() so the execution ain't made simultaneously between the children.

waitpid(id-proc1, NULL, 0);
waitpid(id-proc2, NULL, 0);

`WAITPID` → Instead of waiting for whatever child process to finish execution, it's going to wait for a specific child process to finish exec.

first param → VAR's name

second param → pointer to return the status

third param → option = 0 → end of any process from the same group who calls
-1 → end of every child process

and some more options ---

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

path → the path to our command

argv → the args the command needs / if command has options need to use 'split' = argv = { "ls", "-la", null } need to be null terminated

envp → the environmental variable

↳ Dans le parsing

char *envp-path;
char **path;
char **arg;

envp-path = ft_substr(envp...); // retrieve the line
path = ft_split(envp-path, ":"); // path from envp.
↳ add "/" at the end to work properly

arg = ft_split(argv[2], " ");

↳ In the child process

int i; char *cmd,

i = -1;

while (path[i])

{ cmd = ft_join(path[i], arg[2]); protect!
execve(cmd, arg, envp); // if error ret -1

free(cmd);

ACCESS → function that is checking if the command (input) exists before its execution.

int access(const char *path_name, int a-mode);

pathname → indique le fichier à check.

a-mode → mode d'accès à tester = F_OK
testé l'existence du file ↴
- teste permission exec = X_OK

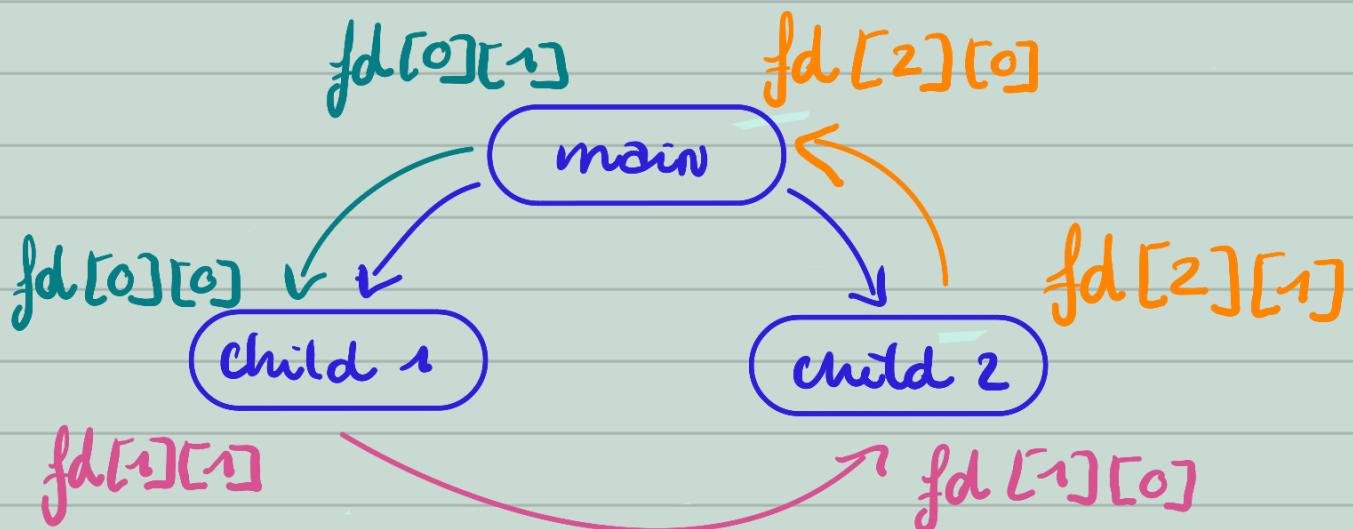
R_OK permission de lecture

W_OK permission d'écriture

Working with multiple pipes

→ if we want for example 3 pipes for 3 commands
ON crée les pipe avec une boucle.

↳ int fd [3][2]
index of pipe ↑ file descriptor



- ensuite on fork pour chaque processus enfant.
- On close tout ce qui n'est pas utilisé.
- On execute les commandes
- On close ce que l'on n'a pas utilisé.

main : → initialisation par défaut
→ compte le nombre de cmd
→ parser les commandes et path
→ parser infile et outfile
→ initialiser et malloc le nombre de fd nécessaires

pipex : → fork le nombre de fois nécessaire
↳ protéger
↳ entrer dans le processus

↳ gérer la première commande
↳ parser les files / et ouvrir dup2 pour input et fd exécuter
↳ gérer le reste des cmd's une à la x
↳ parser les files / et ouvrir dup2 pour output et fd exécuter

→ close tous les fd de chaque processus

→ close l'infile et l'outfile
l'input l'output

→ wait pour chaque exit
status de chaque process

execution = → close tous les fd

→ check cmd access

→ executer

↳ si erreurs print message

↳ close tant

↳ free tant

↳ exit (exit-status)

free → free double pointeur

→ ! toujours remettre à NULL

→ free pipe

