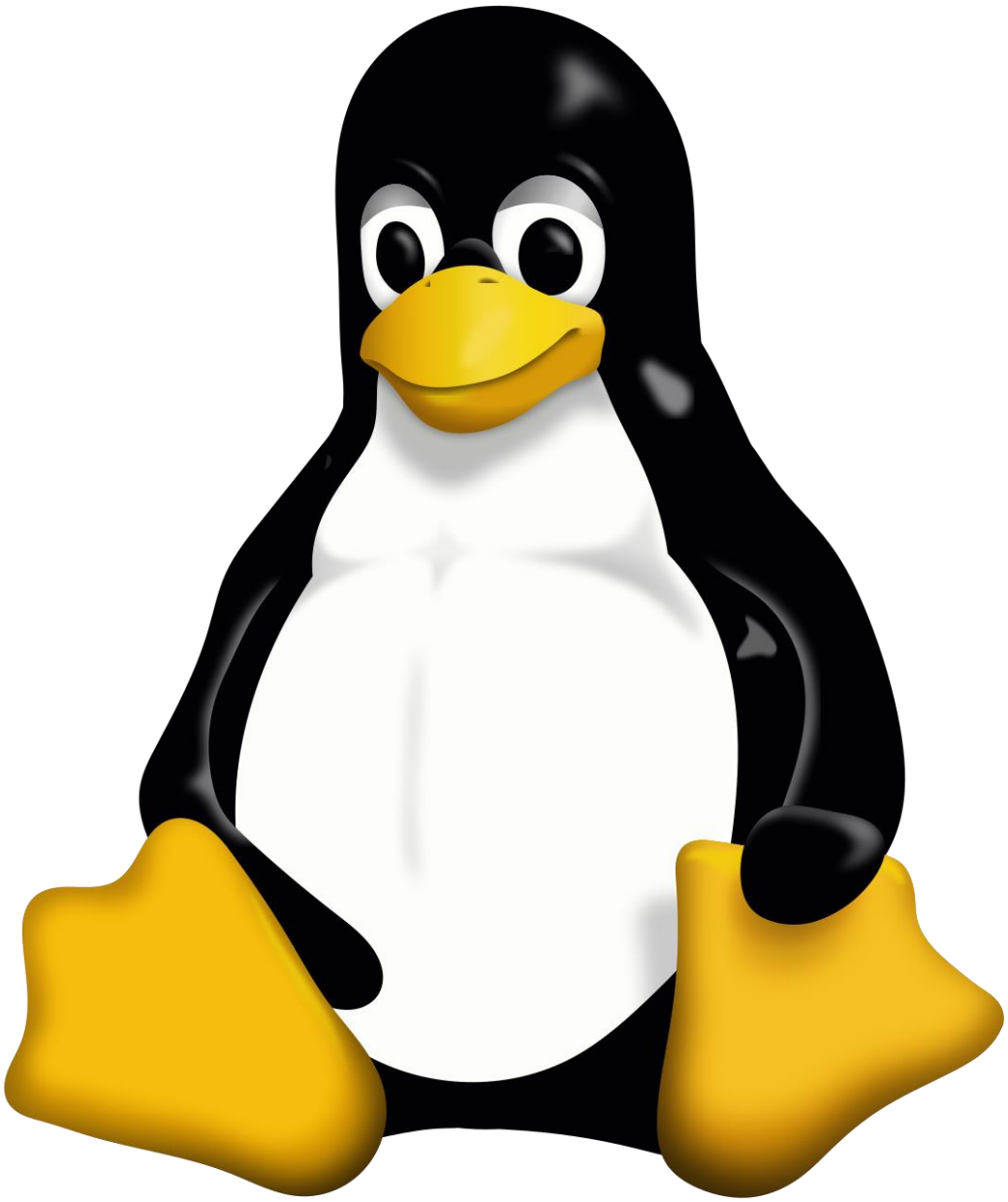
DON’T COMPROMISE



MINI SHELL

By:

ELASRI Mohammed

UNIVERSITÉ Grenoble Alpes

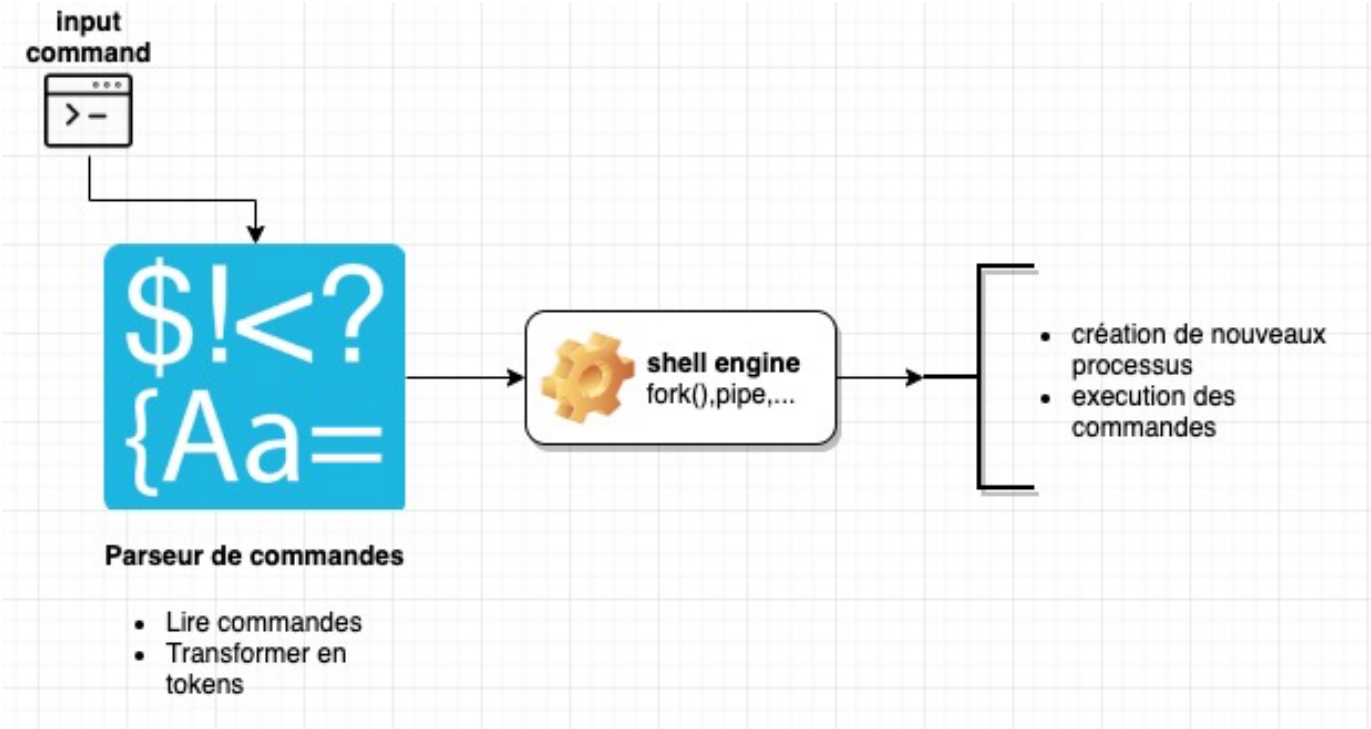
TABLE DES MATIÈRES

1. PRÉSENTATION DE PROJET

Mini SHELL est un projet qui se porte sur la programmation ou une simulation du SHELL utilisé sur Linux en se basant seulement sur quelques fonctionnalités de base.

Au début on vous présentera une petite conception qui trace le chemin à partir laquelle on peut atteindre chaque fonctionnalité de notre Shell.

2. CONCEPTION GLOBAL

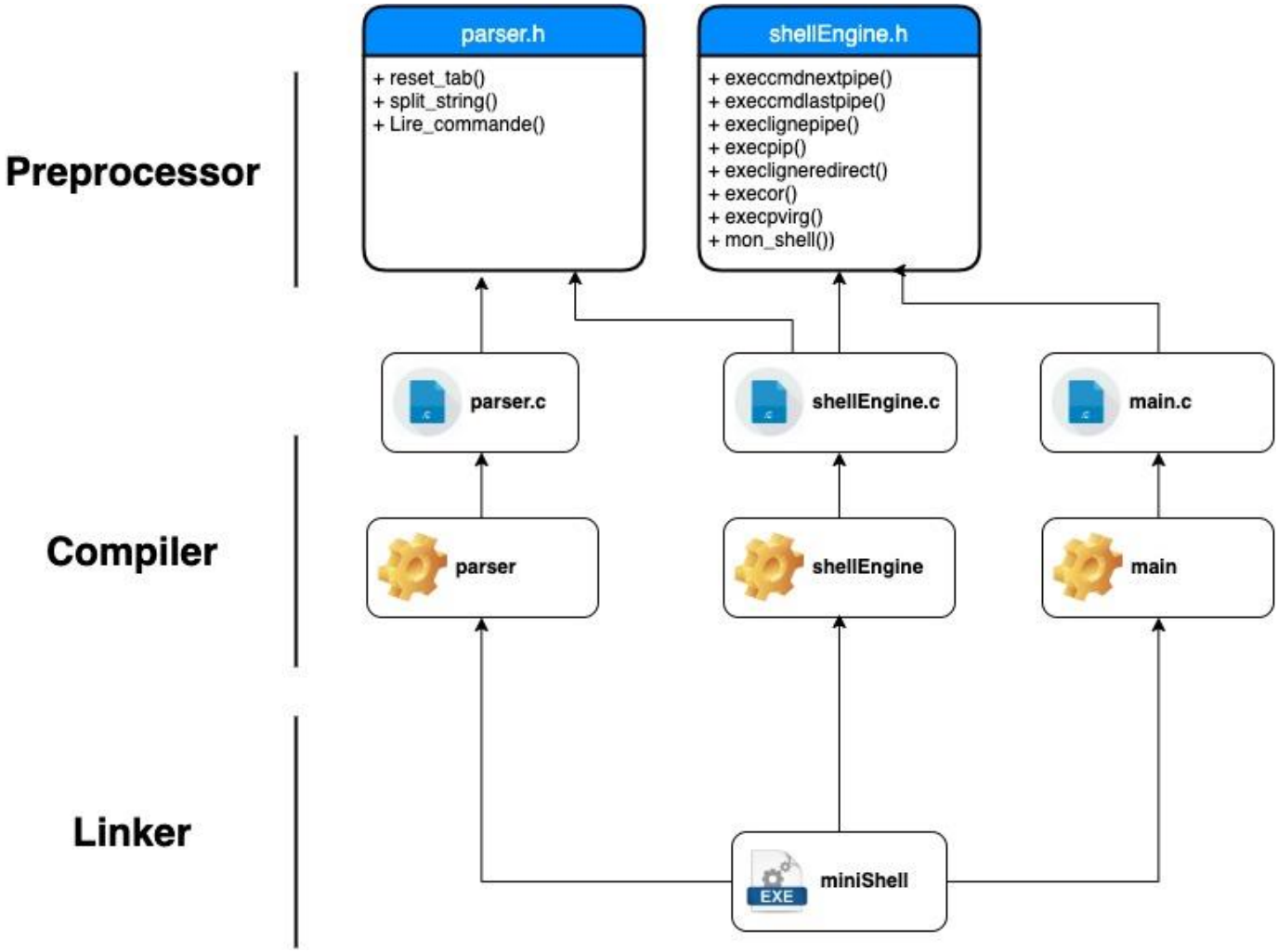


L'utilisateur est censé d'entrer une commande tant que notre Shell est en attend d'une entré sinon il peut quitter en utilisant la commande « Q »
Il y a une partie qui s'occupe de parsing de la commande avant de l'envoyer vers le core de note Shell qui contient des fonctions qui traite la commande et créer des processus en dépendant de sa forme ..

3. Structure des fichiers et Makefile

Comme mentionné précédemment dans la conception, c'est évident que la définition de nos fonctionnes principales se situe dans le **shellEngine**, la définition des fonctions pour le parsing se situe dans **parsing**. Pour arriver à notre exécutable final, on passe par trois étapes

- **Préprocesseur** : le préprocesseur s'occuper des directives, il leurs remplace par leur contenu référencé pour produire des fichiers purement de c.
- **Compiler :** le compilateur va produire des objets à partir de fichier résultat de préprocesseur
- **Linker :** attacher notre programme aux bibliothèques standards utilisées avant de produire un exécutable



En se basant sur la structure de fichiers précédente, on va produire un fichier MakeFile qui va automatiser les étapes qu'on a décrit jusqu'à la phase finale, c'est l'exécutable !

```
CC=gcc
CFLAGS=
LDFLAGS=
EXEC=miniShell

all: $(EXEC)

miniShell: miniShell shellEngine.o parser.o main.o
    $(CC) -o miniShell shellEngine.o parser.o main.o

parser.o: parser.c
        $(CC) -o parser.o -c parser.c $(CFLAGS)

shellEngine.o: shellEngine.c parser.h
    $(CC) -o shellEngine.o -c shellEngine.c $(CFLAGS)

main.o: main.c shellEngine.h
    $(CC) -o main.o -c main.c $(CFLAGS)


run :
    ./$(EXEC)

clean: clean
    rm -rf $(EXEC)
    rm -rf *.o

```

Don pour tester notre **miniShell** on peut procéder avec les commandes suivantes :

➔ Compilation du projet:
- make clean
- make

➔ Exécution du projet :
- make run

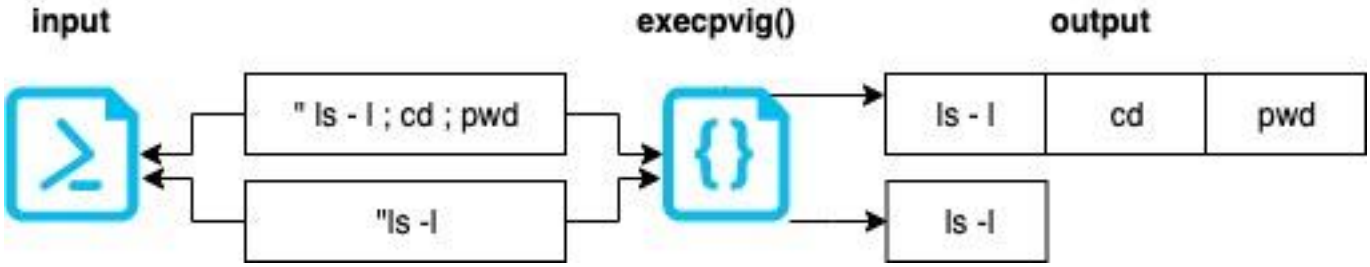
4. Structure des fichiers et Makefile

I. Traitement des commandes séparé par point-virgule

Le core de notre Shell passe toujours par la méthode **mon_shell (),**
Cette dernière va répéter à demander et attendre une entrée de la forme : chaine de caractère qui doit être une commande valide ou « Q » pour quitter.

Dès que l'utilisateur entre une commande on fait l'appel **execPvig ()** sont travail est de voir si notre commande est bien repartie en sous-commandes séparé par points-virgules, à ce stage c'est la fonction **Lire_commande** () du module **parser** qui va diviser la chaine de caractères {la commande}.
On aura deux l'un des deux résultats :

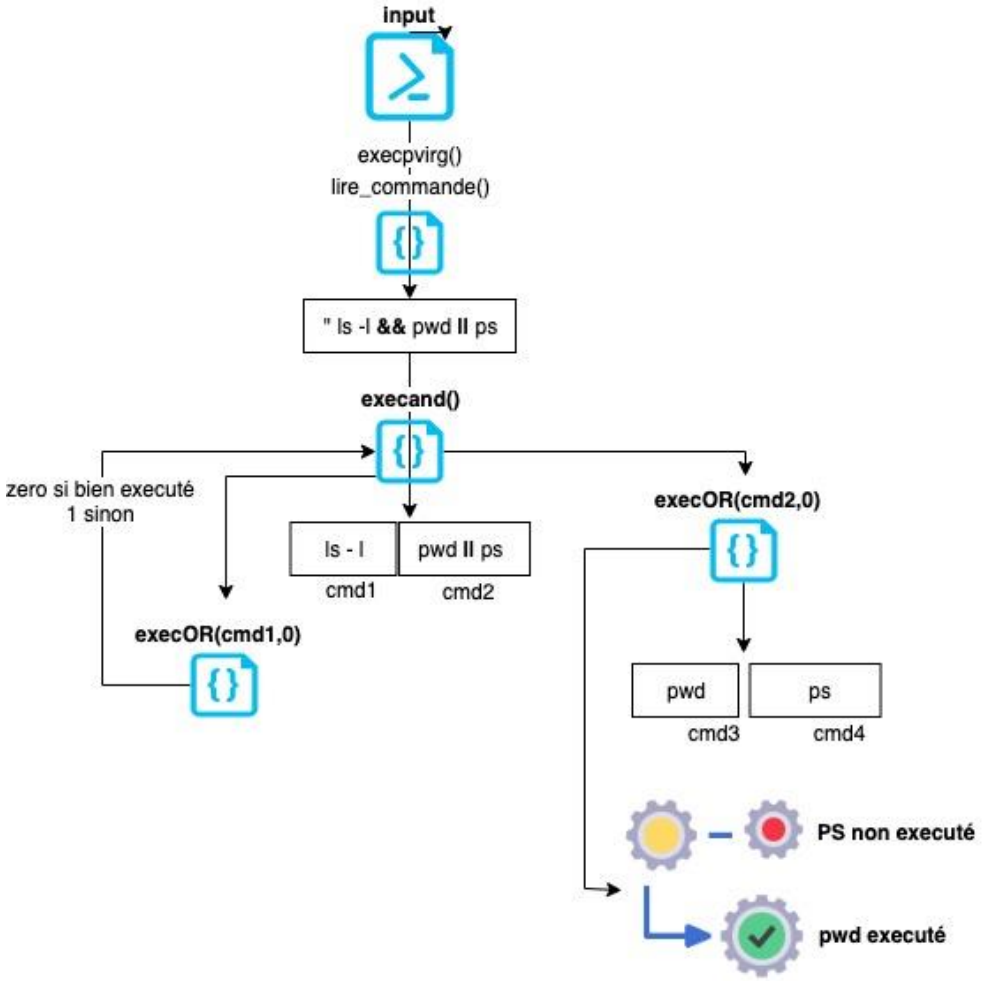
- ◊ Un tableau des sous chaines
- ◊ La commande originale au cas où il n'y a pas de point-virgule.



II. Traitement de AND et OR

Le traitement de AND se fait à travers la fonctionne **execAND (),** cette dernière va être s'exécuté plusieurs fois par **execPvig ()** au cas où notre commande contient des points virgules, sinon une seul fois.
En fait le traitement du AND et OR se faite horizontalement,
ExecAND () va vérifier si notre commande contient un « && », si oui on va prend la première partie qui précède le AND l'évaluer si elle est exécutée sans erreur, on envoie un « 0 » pour que la commande juste après le AND s'exécute par la fonction **execOR** ().
S'il y a une erreur sur la première partie on envoie « 1 » pour que **execOR** ignore la première commande.

IV. Traitement de AND et OR :



Dans cette figure on veut exécuter la commande **« ls -l && pwd || ps » .**
La fonctionne execAND sépare notre commande on deux commandes **cmd1** et **cmd2**, la fonctionne **execOR** () va encore diviser **cmd2** en deux **cmd3** et **cmd4** car il contient un **« || » .**
Chaque fois **execOR** est exécuté elle reçoive un paramètre en plus de la **cmd**.
Pour la première fois on l'envoie un zéro pour lui dire exécute **cm1.**
On évalue son résultat, si la commande renvoie zéro ça veut dire il était bien exécuté dans notre cas on va exécuter **cmd2** car **« ls -l »** marche bien.
Maintenant après reçu **cmd2** par **execOR,** on va exécuter **cm3** et puisque **pwd** marche bien on va ignorer la commande **Ps**.

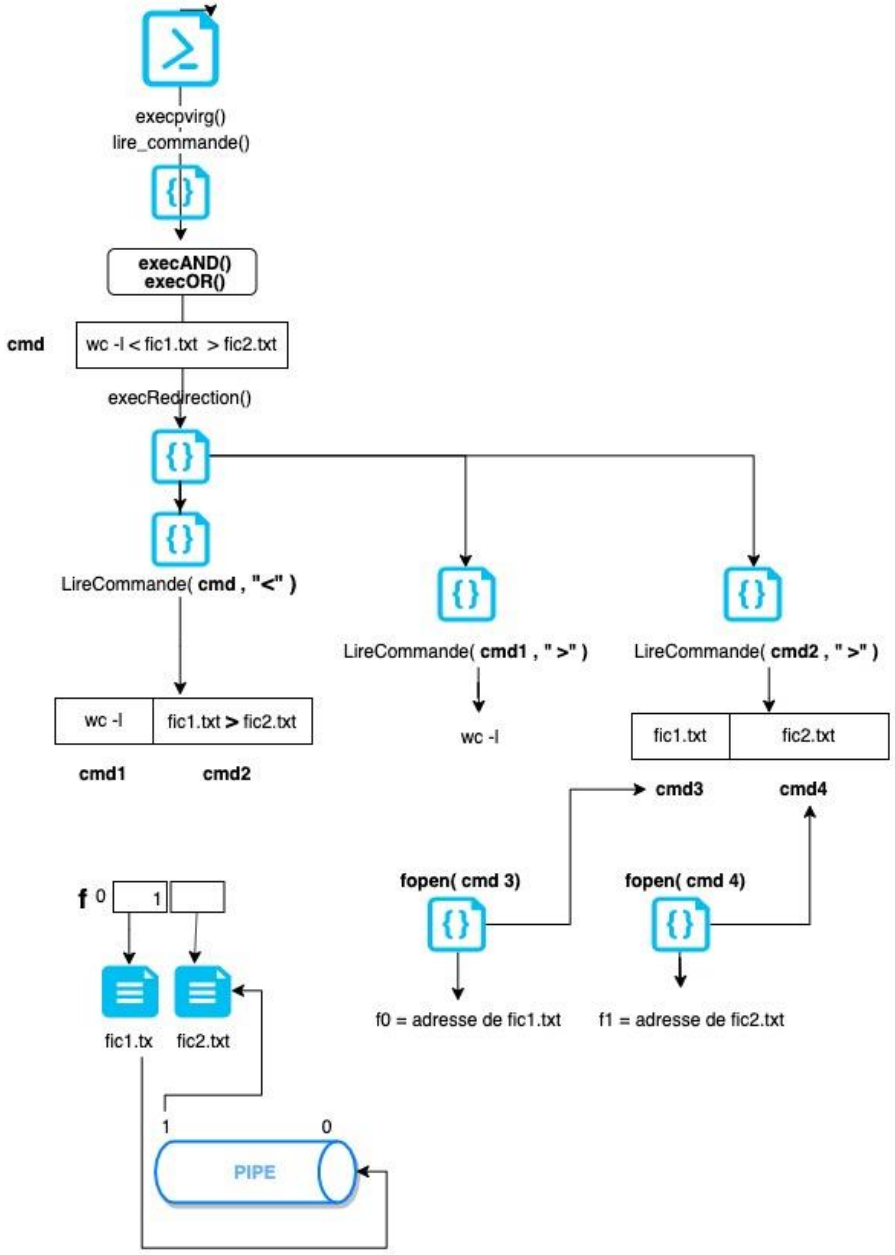
Difficulté Rencontré :

Lors de construction des deux fonctions **execAnd** () et **execOR** ()
On a trouvé un problème pour exécuter les commandes (contenant AND et OR) séquentiellement et avec le même niveau de priorité.

Solution Trouvé :

L'ajoute d'un paramètre à la fonctionne **execOR** () à chaque fois on l'appel. comme mentionné dans la figure précèdent.

III. Traitement des redirections

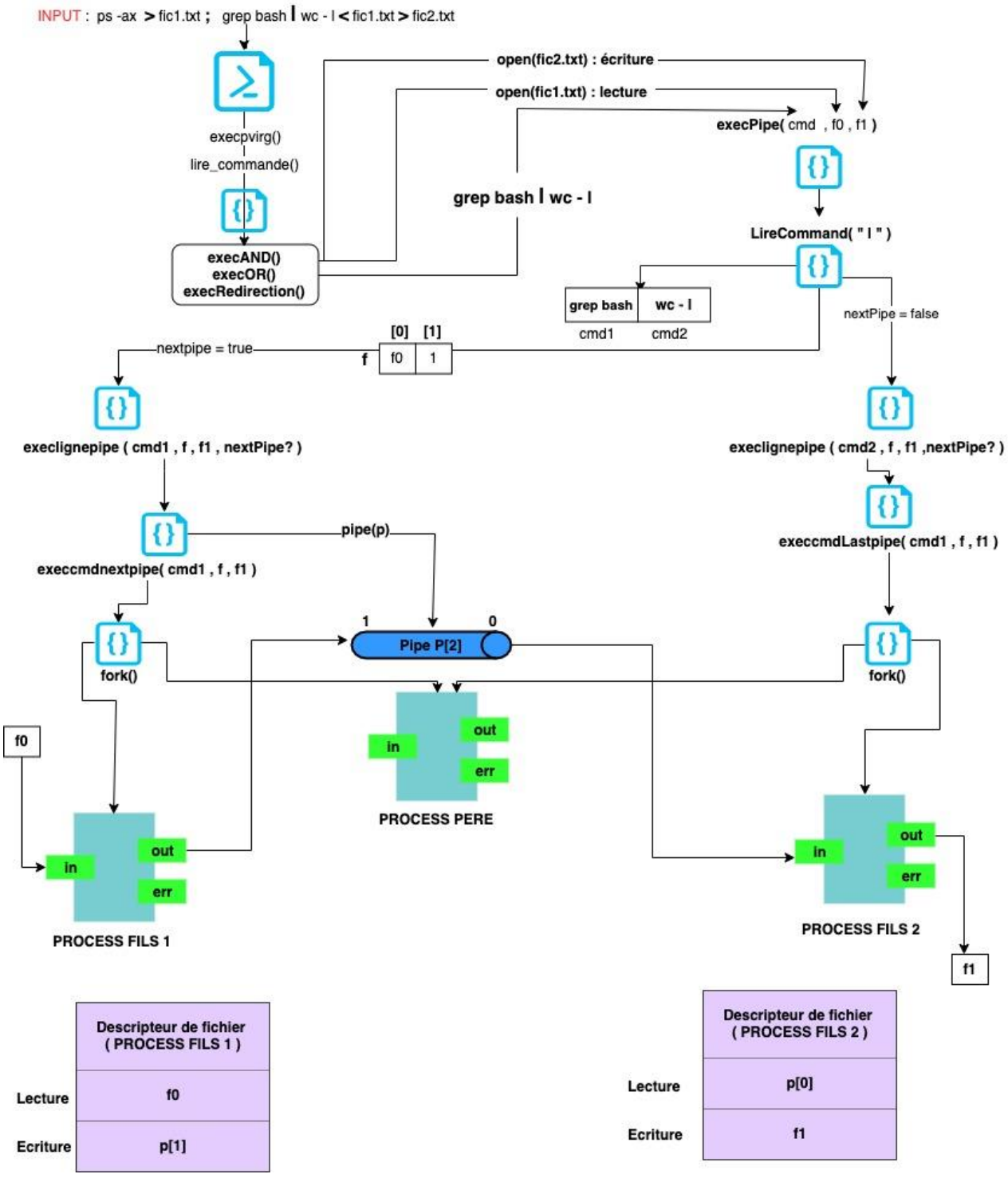


La figure précédente illustre le traitement de la pipe par notre Shell.
le traitement se fait par la fonction **execRedirection** (), cette fonction traite notre commande **cmd** et y converti en sous chaines **cmd1** et **cmd2** car il sont séparé par redirection à gauche (lecture), ensuite on va traiter chaque sous commande cette fois avec la redirection à droite « > » ce qui dans en résultat une commande de base « wc -l » qu'on va exécuter prochainement comme command externe , et deux commande **cmd3** et **cmd4 .** Ces deux dernières commandes contiennent le nom du fichier d'écriture **cmd4** (vu que « > » est forcément suivit par un nom de fichier), par le même principe on récupère le nom du fichier de lecture depuis **cmd2.**
On va les traiter avec **open** () pour ouvrir ces deux fichiers et récupérer leurs adresses. Ces adresses on va les stocker dans la variable **f0** pour lecture et **f1** pour l'écriture, ensuite on va utiliser ces deux variables comme entrée et sortie de l'exécution de la fonction **execpipe** ().
On va aborder ça dans la partie prochaine.

Note : f0 et f1 sont initialisé avec la valeur 0 ➔ s'il n'y a pas de redirection « < ou > », f0 et f1 auront la valeur 0.

V. Traitement des pipes :

L'utilisation des pipes dans notre Shell couvre tous les cas, on peut traiter un nombre infini des pipes et on y combinant avec plusieurs commandes (redirection, OR et AND, redirection …).



Après que notre commande passe par **execpvirg** () elle va se diviser en sous commandes, chaque sous commande va être traité par les fonctions **execAnd** () et **execOR** (), ensuite la commande va être traité par **execRedirection** ().
cette dernière va appeler **execPipe (cmd, f0 ,f1)** qui cherche les pipes et traite ses commandes séparément en appelant **execLinePipe**(**cmd** , **f**, **f1, nextPipe?**) ;
Cmd : c'est la commande de base à exécuter.
f : c'est un tableau [2] qui la valeur(f[0]) à mettre en lecture de la commande**.**
(Close (0) ; dup(f[0]) ;) .
f1 : c'est un entier qui contient la valeur à mettre en écriture dans la commande après le dernier pipe.
NextPipe : c'est un booléen qui indique s'il y a un pipe après la commande.
ExecLinePipe : cherche si la commande à exécuter c'est une commande interne à ce moment-là, elle l'exécute en commande interne, sinon elle vérifie **NextPipe?**
si **NextPipe** = true alors elle exécute **execCmdNextPipe(cmd,f,f1)**
Sinon , elle execute **execCmdLastPipe(cmd,f,f1).**
la fonction **execCmdNextPipe** crée un nouveau pipe **P** et fait un fork .

- Dans le processus fils ont met en lecture **f[0]** et en écriture **P1** et puis on execute la commande.
- Dans le processus père on ferme **f** et on modifie le tableau **f** avec les valeurs du nouveau pipe, f[0] = p[0] ; f[1] = p[1] ;

La fonction **execCmdLastPipe** fait un fork .

- Dans le processus fils on met en lecture f0 et en écriture f1 et on execute la commande.

Après avoir connecté toutes les commandes par des pipes on fait une boucle while (**waitpid() !=-1)** pour attendre l'exécution de toutes processus fils.
(On verra plus tard dans la partie l'exécution en background pourquoi on utilise **waitpid()** et non **wait()**)
Si une des commande s'est mal exécuter on retourne 1 sinon on retourne 0.

IV. Exécution en background :

Pour l'exécution en background des commandes malheureusement par manque de temps on a pas pu l'implémenter en commande combiner avec les pipes redirection …
Par contre nous avons pu l'implémenter en commander simple comme emacs &
Notre fonction **execbg(cmd)** recherche dans la ligne de commande cmd l'existence de « & » sans confondre avec « && » et ensuite s'il est trouvé notre fonction exécute la commande mais avant elle change le group id avec la fonction **setpgid()** et on lui donne en paramètre le pid du processus fils pour le séparer du processus père et on fait pas un wait() ce qui permettra a la commande de s'exécuter en background.
Dans la fonction **execpip** nous avons utilisé **waitpid()** et on lui a donner en premier paramètre 0 pour que le wait se fait sur tous les processus fils qui appartient au même group id du père et vu que dans **execbg()** nous avons séparé le fils du père waitpid ne va jamais attendre le processus exécuter en background.

V. Interception de signaux :

Pour l'interception des signaux nous avons utilisé la fonction **signal()** et on lui a donnée en paramètre soit **SIGQUIT** ou **SIGINT** en fonction du signal qu'on souhaite intercepter et exécuter respectivement les fonction **stopexecquit** et **stopexecint**.
Stopexecquit() vérifie si le processeur qui l'appel est le père ou pas
Si le processeur c'est un fils alors à ce moment-là elle exit(0) sinon elle ne fait rien
Même fonctionnement pour **stopexecint.**

VI. Exemple de commande prise en charge :

Ps ax > test.txt
Grep bash | wc –l < test.txt > test2.txt
Commande inexistante || ps ax | Grep bash | wc –l && ls > test.txt
Commande inexistante ; ps pax ; cd .. ; setenv
Emacs &

VII. Sources

Dans notre programme nous avons récupérer la fonction **split_string** du site stackoverflow.com plus précisément sur le poste suivant :
https://stackoverflow.com/questions/34675966/strtok-when-the-delimiter-is-string
La fonction elle fait le même fonctionnement que strtok sauf qu'elle coupe avec la chaine de caractère passer en paramètre contrairement a strtok qui tok la chaine de caractère avec tous les char qui existe dans le 2eme paramètre de la fonction
Différence de comportement entre **strtock** et **split_string :**

Pour un appel en boucle sur **strtock(**« ps ax | wc –l || ls » , « || »**)** la fonction va découper comme suit :

ps ax	wc -l	ls

Alors que la fonction **split_string(**« ps ax | wc –l || ls » , « || »**)**

ps ax | wc –l	ls

Fonctionnalité	Réalisée (oui/non)	Nom de fonction,	nom de fichier
Commandes internes	OUI	execLignPipe()	shellEngine.c
Redirection d'E/S	OUI	ExecLigneRedirect()	shellEngine.c
Tubes	OUI	execPipe ()	shellEngine.c
Enchaînement de commandes	OUI	Execpvirg() ExecOR() ExecAND()	shellEngine.c
Mise en background	OUI	Execbg()	shellEngine.c
Interception de signaux	OUI	stopexecquit() stopexecint()	shellEngine.c
Autres :	OUI	Lire_commande() split_string	Parser.c