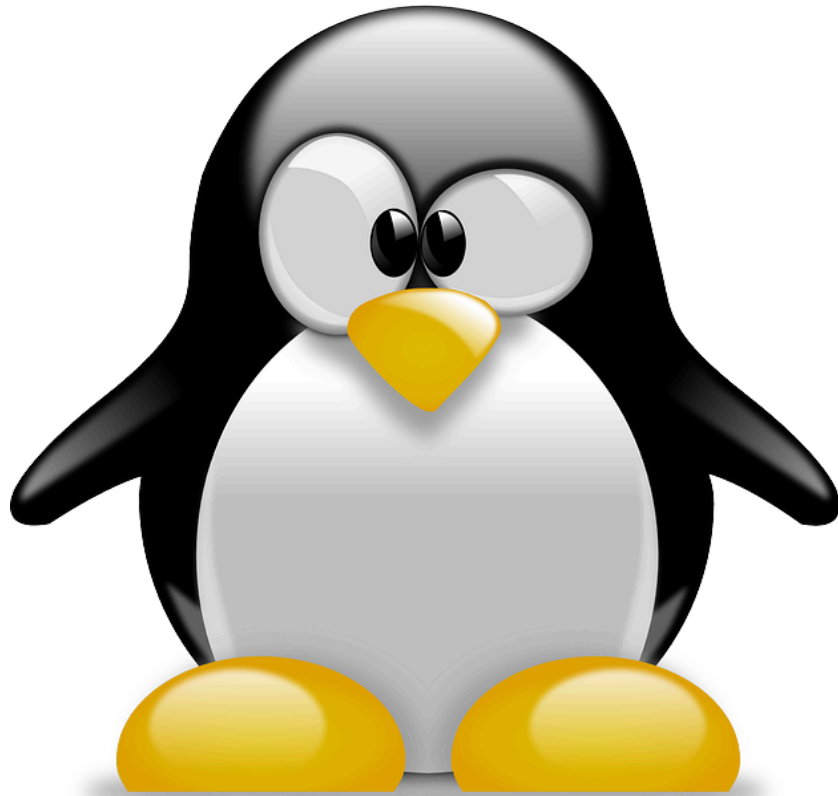


Fusi Belaid
Thomas
4A

Projet 1
ProgSys



1. Introduction	3
2. Consignes	3
3. Code du minishell	4
4. Tests	5
5. Conclusion	6

1. Introduction

Ce projet de Programmation Système vise à renforcer les concepts fondamentaux de la gestion des processus et des appels système en environnement Unix. Dans cette première partie, nous avons développé un minishell en Python, une version simplifiée d'un interpréteur de commandes. L'objectif principal était de permettre l'exécution de commandes externes à partir de la ligne de commande, en utilisant des fonctions de la famille `os.exec`. À travers ce projet, nous avons traité la gestion des processus en avant-plan et en arrière-plan, tout en prenant soin de gérer les erreurs potentielles via des exceptions Python (`OSError`). Ce travail constitue une exploration pratique des mécanismes sous-jacents à l'exécution des commandes dans un système Unix/Linux, sans les fonctionnalités avancées de redirection ou de combinaison de commandes.

2. Consignes

Le projet est d'écrire un shell simplifié en python.

Écrire un programme Python (*myshell.py*) qui lit des lignes de commande à l'entrée standard et les lancent en exécution, à l'instar du *shell* standard Linux. Le programme ne doit pas utiliser la fonction *system*, l'exécution est lancée avec une fonction de la famille *os.exec*.

Pour simplifier le travail, nous allons faire les hypothèses suivantes :

- Une commande est formée d'une seule ligne. Seules les commandes *externes* (i.e., programmes exécutables avec arguments) seront traitées.
- Les arguments sont séparés par des espaces. Les caractères d'échappement (`\`), guillemets, etc. ne sont pas traités.
- Les opérateurs de combinaison de commandes (`|`, `||`, `&&`, etc.) ou de redirection (`<`, `>`, etc.) ne sont pas pris en compte.
- En revanche, la ligne **peut se terminer par &** (séparé du dernier argument par un espace) ce qui produit un effet similaire au lancement en arrière-plan en *shell* standard.
- A la fin d'un processus lancé en premier plan (sans `&`), le shell affiche son PID et son code de retour

*A noter : il convient de traiter les erreurs potentielles à chaque appel système (exception **OSError**) et afficher une description de la cause de l'erreur (cf. **OSError.strerror**)*

3. Code du minishell

Voici le code python du minishell

```
import os

cmd = input("$ ")
while cmd != "quit":
    args = cmd.split()
    if args[-1] == "&":
        args = args[:-1]
        background = True
    else:
        background = False

    pid = os.fork()
    if pid == 0: #enfant
        try:
            print(f"args: {args},    args[0]: {args[0]}")
            os.execvp(args[0], args)
        except OSError as e:
            print(f"Erreur d'exécution de la commande: {e.strerror}")
            os._exit(1)

    else: #parent
        if not background:
            status = os.waitpid(pid, 0)[1]
            print(f"Commande {cmd} executée premier plan avec le PID: {pid}, Code de retour: {os.WEXITSTATUS(status)}")
        else:
            print(f"Commande en arrière-plan avec PID: {pid}")

    cmd = input("$ ")
```

4. Tests

- 1) Test avec la commande ls sans argument et en premier plan

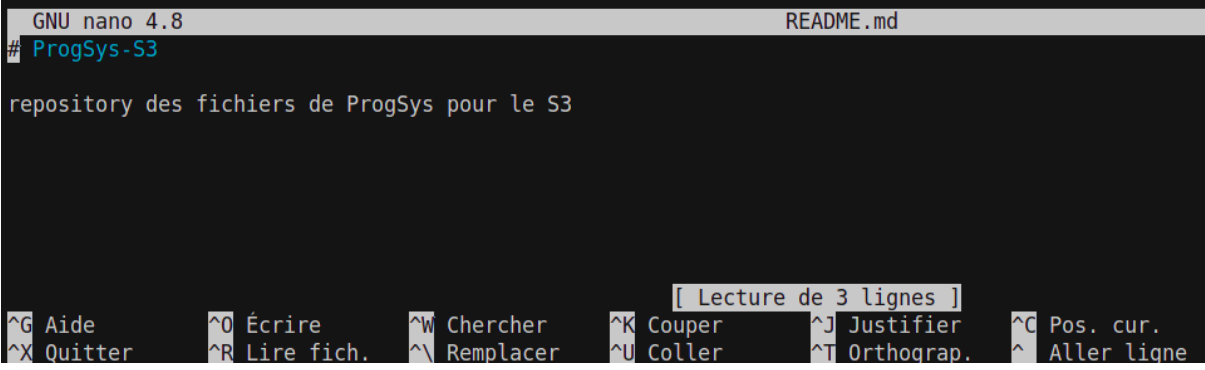
```
$ ls
args: ['ls'], args[0]: ls
Projets README.md sortie.txt TD TP
Commande ls exécutée premier plan avec le PID: 8745, Code de retour: 0
```

- 2) Test avec la commande ls au premier plan avec des arguments

```
$ ls -l /
args: ['ls', '-l', '/'], args[0]: ls
total 2097232
lrwxrwxrwx 1 root root 7 sept. 6 17:20 bin -> usr/bin
drwxr-xr-x 4 root root 4096 sept. 6 15:36 boot
drwxrwxr-x 2 root root 4096 sept. 6 17:24 cdrom
drwxr-xr-x 20 root root 4800 oct. 11 15:58 dev
drwxr-xr-x 137 root root 12288 sept. 24 16:46 etc
drwxr-xr-x 3 root root 4096 sept. 6 17:24 home
```

- 3) Test avec la commande nano

```
$ nano README.md
```



```
GNU nano 4.8 README.md
# ProgSys-S3
repository des fichiers de ProgSys pour le S3

[ Lecture de 3 lignes ]
^G Aide      ^O Écrire    ^W Chercher  ^K Couper    ^J Justifier  ^C Pos. cur.
^X Quitter   ^R Lire fich.^_ Remplacer  ^U Coller    ^T Orthograp.^_ Aller ligne

$ nano README.md
args: ['nano', 'README.md'], args[0]: nano
Commande nano README.md exécutée premier plan avec le PID: 9369, Code de retour: 0
```

- 4) Test avec la commande xeyes au 1er plan
Les tests fonctionnent mais je ne peux pas prendre de capture d'écran
- 5) Test avec la commande ls en arrière plan

```
$ ls -l / &  
Commande en arrière-plan avec PID: 9861  
$ args: ['ls', '-l', '/'], args[0]: ls  
total 2097232  
lrwxrwxrwx    1 root root          7 sept.  6 17:20 bin -> usr/bin  
drwxr-xr-x    4 root root     4096 sept.  6 15:36 boot  
drwxrwxr-x    2 root root     4096 sept.  6 17:24 cdrom
```

5. Conclusion

Ce projet nous a permis de consolider notre compréhension des appels système, en particulier l'exécution de processus via les fonctions `os.exec` de Python. En développant ce minishell, nous avons appris à manipuler les processus, à gérer les exécutions en arrière-plan, et à traiter les erreurs avec précision pour assurer la robustesse de notre shell simplifié. Cette expérience nous a également permis de découvrir les limitations et les particularités d'un shell minimaliste par rapport à un shell complet, tel que Bash. Les compétences acquises ici seront utiles pour les prochaines parties du projet, où nous explorerons des fonctionnalités plus avancées de la programmation système.