Systèmes d'exploitation centralisés

Rapport : minishell

1SN-F

Élève :

Louis Thevenet

Enseignant :

Emannuel Chaput

4 Mai 2024

Table des matières

1. Gestion des processus 3
   1.1. Enchaînement séquentiel des commandes 3
   1.2. Exécution en arrière-plan 3
2. Gestion des tâches 3
3. Signaux 4
   3.1. Signal `SIGCHLD` 4
   3.2. Signaux `SIGINT`, `SIGTSTP` 5
4. Fichiers et redirections 6
5. Tubes 7

1. Gestion des processus

1.1. Enchaînement séquentiel des commandes

```
> ls
flake.lock  flake.nix  projet  rapport  result  sujets  tp
> echo test
test
> cat ./projet/test
exemple
> exit
Au revoir ...
```

Liste 1. – On attend la fin de l'éxecution du fils pour passer à la prochaine commande

1.2. Exécution en arrière-plan

```
> cat ./projet/test
#!/usr/bin/env bash

sleep 5
echo "Done!"
> ./projet/test &
>
> ls
flake.lock  flake.nix  projet  rapport  result  sujets  tp
> Done!
>
```

Liste 2. – On exécute la commande en arrière-plan

On peut également vérifier la bonne terminaison du fils après éxecution via `watch ps -sf` :

```
S+   pts/2      0:00      \_ /nix/store/cci0aml5v6xdvkqrvg-minishell/bin/minishell
S+   pts/2      0:00          \_ bash ./projet/test
S+   pts/2      0:00              \_ sleep 5
```

Liste 3. – Durant l'exécution

```
S+   pts/2      0:00      \_ /nix/store/cci0aml5v6xdvkqrvg-minishell/bin/minishell
```

Liste 4. – Après exécution

2. Gestion des tâches

Comme proposé dans l'énoncé du projet, on ajoute des commandes internes au minishell :
lj (list jobs) Affiche les tâches en cours
sj <id> (stop job) Arrête la tâche d'identifiant `id` (envoie `SIGSTOP`)
fg <id> (foreground) Met la tâche d'identifiant `id` en avant-plan
bg <id> (background) Met la tâche d'identifiant `id` en arrière-plan

Ces commandes seront utilisées dans la suite pour illustrer l'état du minishell.

De plus, un mode `debug` a été ajouté au projet afin d'afficher des informations sur les signaux reçus. Il fournit des messages de logs durant le foncitonnement.

Pour l'activer, il faut compiler en définissant `DEBUG` qui change le comportant de la macro suivante :

```c
// #define DEBUG // A DECOMMENTER

#ifdef DEBUG
#define LIGHT_GRAY "\033[1;30m"
#define NC "\033[0m"
#define DEBUG_PRINT(x)                                                         \
  printf("%s", LIGHT_GRAY);                                                   \
  printf x;                                                                    \
  printf("%s", NC);
#else
#define DEBUG_PRINT(x)                                                         \
  do {                                                                         \
  } while (0)
#endif
```

3. Signaux

3.1. Signal `SIGCHLD`

Dans la Liste 5, on :

- attend normalement la fin d'éxecution de la commande en avant-plan
- attend normalement la fin d'éxecution de la commande en arrière-plan
- envoie le signal `SIGCHLD` au processus fils
- envoie le signal `SIGSTOP` au processus fils
- envoie le signal `SIGCONT` au processus fils
- affiche les jobs en cours pour constater que le fils est continué en arrière-plan
- On teste ensuite les signaux `SIGSTOP` et `SIGCONT` sur un job en arrière-plan.

```
> sleep 2
[Child 294501 exited]
> sleep 2&
> [Child 294709 exited]

> sleep 9999
[Child 296018 signaled]
> sleep 9999
[Child 299625 stopped]
> [Child 299625 continued]

> lj
Job 0
  PID: 299625
STDOUT -> 4207432  State: Active  Command: sleep 9999
>
>
>
>
>
> sleep 999&
> [Child 335139 stopped]
[Child 335139 continued]

> lj
Job 0
  PID: 299625
STDOUT -> 4207432  State: Active  Command: sleep 9999

Job 1
  PID: 335139
STDOUT -> 4207432  State: Active  Command: sleep 999
>
```

Liste 5. – Démonstration `SIGCHLD`

3.2. Signaux `SIGINT`, `SIGTSTP`

On voit dans cet exemple que le programme père reçoit le signal `SIGINT`, qu'il décide de tuer le fils en avant-plan, finalement le message informant la terminaison du processus fils est affiché.

```
> sleep 10
^C[SIGINT received]
[Killing 343628]
> [Child 343628 signaled]

>
```

Liste 6. – Interruption au clavier

```
> sleep 10
^Z[SIGTSTP received]
[Stopping 348897]
> [Child 348897 stopped]

> lj
Job 0
  PID: 348897
STDOUT -> 4207432  State: Suspended  Command: sleep 10
>
```

Liste 7. – Envoie du signal `SIGTSTP`

Les processus fils quant à eux ne reçoivent pas les signaux `SIGINT` et `SIGTSTP`, c'est le père qui gère les interruptions.

```
> sleep 999&
> ^C[SIGINT received]
^C[SIGINT received]
^Z[SIGTSTP received]
^Z[SIGTSTP received]

> lj
Job 0
  PID: 355672
STDOUT -> 4207432  State: Active  Command: sleep 999
>
```

Liste 8. – Les processus fils masquent les signaux `SIGINT` et `SIGTSTP`

4. Fichiers et redirections

Le mode debug affiche les nombres d'octets lus et écrits.

```
> cat < ./projet/test
[read 42, wrote 42]
[read 0, wrote 0]
[Child 137265 exited]
#!/usr/bin/env bash

sleep 5
echo "Done!"
[Child 137264 exited]
> cat < ./projet/test > test
[read 42, wrote 42]
[read 0, wrote 0]
[Child 137384 exited]
[read 42, wrote 42]
[read 0, wrote 0]
[Child 137383 exited]
> [Child 137385 exited]

> cat test
#!/usr/bin/env bash

sleep 5
echo "Done!"
[Child 137434 exited]
```

Liste 9. – Démonstration des redirections vers des fichiers en avant plan

L'exemple suivant montre que les redirections continuent de fonctionner en arrière-plan et respectent l'état du processus qui les envoit ou reçoit. On lance en arrière-plan un script dont la sortie est redirigée vers un fichier, on peut le mettre en pause et le reprendre sans que la redirection ne s'arrête.

```
> cat ./projet/test_boucle
#!/usr/bin/env bash

for i in {1..50}; do
  echo "Iteration $i"
  sleep 3
done
echo "Done!"
> ./projet/test_boucle > sortie &
> cat sortie
Iteration 1
Iteration 2
> #on attend un peu
Error: command failed to execute
> cat sortie
Iteration 1
Iteration 2
Iteration 3
Iteration 4
> lj
Job 0
  PID: 153453
STDOUT -> 4  State: Active  Command: ./projet/test_boucle
> sj 0
> lj
Job 0
  PID: 153453
STDOUT -> 4  State: Suspended  Command: ./projet/test_boucle
> cat sortie
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
> cat sortie
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
> bg 0
> cat sortie
Iteration 1
Iteration 2
Iteration 3
Iteration 4
Iteration 5
Iteration 6
Iteration 7
Iteration 8
Iteration 9
```

Liste 10. – Démonstration des redirections vers des fichiers en arrière-plan (sans affichage debug)

5. Tubes

```
> ls | wc
[Child 176273 exited]
      9       9      62
[Child 176274 exited]
>
> cat flake.nix | grep pkgs | wc -l
[Child 178832 exited]
[Child 178833 exited]
7
[Child 178834 exited]
>
```

Liste 11. – Démonstration de tubes en avant-plan

L'exemple suivant montre que l'information du descripteur de fichier vers lequel la sortie est redirigée est stockée dans la liste des jobs.

```
>  ./projet/test_boucle | wc &
> lj
Job 0
  PID: 179824
STDOUT -> 4207432  State: Active  Command: ./projet/test_boucle
Job 1
  PID: 179825
STDOUT -> 12800864  State: Active  Command: wc
>      11      11      27
[Child 179824 exited]
[Child 179825 exited]
```

Liste 12. – Démonstration de tubes en arrière-plan