Initiation à l'environnement Unix : TP5

novemnre 2024 — Pierre Rousselin

1 Processus

Exercice 1 : Les commandes `ps` et `pstree`

1. Si elle est installée sur votre système, lancez la commande `pstree -T -p`. Que fait-elle ?

a) Quel processus est à la racine de cette arborescence ?

b) Si ce n'est pas déjà fait, ouvrez un navigateur web (et dans ce cas relancez la commande `pstree -T -p`). Combien de processus enfants a-t-il ?

2. À l'aide d'une commande `ps` bien choisie (voir les transparents du cours ou le manuel de `ps`) :

a) Afficher des informations sur le ou les processus de nom de base `systemd` et uniquement celui-là ou ceux-là. Quel est leur utilisateur propriétaire ?

b) Afficher des informations sur les processus dont vous êtes propriétaires et seulement ceux-là.

c) Donner l'ordre de grandeur du nombre total de processus vivant à cet instant sur votre machine à l'aide de la commande `ps -e | wc -l`. Parmi ceux-ci, combien appartiennent à `root` ?

--- * ---

Correction de l'exercice 1 :

1. C'est une commande non standard qui trace l'arborescence des processus. Remarque, l'option `-T` sert à désactiver l'affichage des *threads*, hors programme de ce cours, et l'option `-p` affiche les `PID`.

a) Sur GNU/Linux, la plupart du temps c'est `systemd` qui est le processus numéro 1.

b) A priori une demi-douzaine. Ça dépend du navigateur et sans doute de la machine.

2. a) `ps -f -C systemd` A priori il y a une instance de `systemd` pour `root` et une autre par utilisateur normal connecté à la machine.

b) `ps -f -u $USER`

c) Il y a plusieurs (2, 3, ...) centaines de processus vivants en général. Parmi ceux-ci, pour compter ceux qui appartiennent à `root` : `ps -u root | wc -l`

--- * ---

Exercice 2 : À la recherche de nos ancêtres

1. Afficher le `PID` du shell courant.

2. Donner deux façons d'afficher le `PPID` (c'est-à-dire le `PID` du parent) du shell courant, l'une avec `ps`, l'autre avec un développement de variables.

3. À l'aide de `ps` trouver à quel processus (nom de commande) correspond le parent du shell courant. Chercher le `PPID` de ce processus.

4. Continuer ainsi jusqu'à arriver jusqu'au processus de `PID` 1, de sorte à pouvoir décrire tous les ancêtres du shell courant.

--- * ---

Correction de l'exercice 2 :

1. `echo $$`

2. `echo $PPID` ou bien, par exemple, `ps -f -p $$`.

3. On va sans doute voir dans la ligne ancestrale : `gnome-terminal`, `systemd` pour l'utilisateur et `systemd` pour root. En général on peut systématiser de la façon suivante :

```
$ var=$$; parent=$(ps -o ppid= $var); ps -f -p $parent
$ var=$parent; parent=$(ps -o ppid= $var); ps -f -p $parent
$ var=$parent; parent=$(ps -o ppid= $var); ps -f -p $parent
```

--- * ---

Exercice 3 : Pour rigoler

1. Recopier dans un fichier `lol` le script suivant :
   ```
   #!/bin/sh
   # lol : rigoler après avoir dormi 10 secondes
   sleep 10; echo lol
   ```
   Le rendre exécutable et l'essayer.
2. Recopier dans un fichier `deuxfois` le script suivant :
   ```
   #!/bin/sh
   # deuxfois : exécuter deux fois une commande
   "$@" & "$@"; wait
   ```
   Ce script exécute deux fois la commande formée par ses arguments, se faisant, *fork* deux fois et attend que ses enfants se terminent avec la primitive `wait`. Vous n'avez pas besoin de le comprendre entièrement maintenant.
   Le rendre exécutable et tester la commande `./deuxfois ls /`.
3. Entrer la suite de commandes suivantes :
   ```
   $ pstree -p -T $$
   $ ./deuxfois ./deuxfois ./lol &
   $ pstree -p -T $$
   ```
   Combien de processus ont été créés par la deuxième de ces commandes ?
4. Essayer de prévoir le nombre de `lol` affichés et le nombre de processus créés par la commande
   ```
   $ ./deuxfois ./deuxfois ./deuxfois ./deuxfois ./deuxfois ./lol &
   ```
   Vérifier en visualisant avec la commande `pstree` ci-dessus.

--- * ---

Correction de l'exercice 3 :

1. `chmod u+x lol; ./lol`
2. `chmod u+x ./deuxfois; ./deuxfois ls /` va exécuter deux fois `ls /`, donc lister deux fois le contenu du répertoire racine.
3. La commande `./lol` *fork* elle-même une fois avec `sleep`. Donc chaque `./lol` lancé donnera deux processus. À la fin, cela donne
   $$1 + 2 \times (1 + 2 \times (1 + 1)) = 11$$
   processus.
4. On peut compter le nombre de processus en définissant la suite (u_n) de la façon suivante : $u_0 = 2$ (le nombre de processus créé par `./lol`) et $u_{n+1} = 1 + 2u_n$ (nombre de processus créés par $n + 1$ `./deuxfois` en début de commande). On obtient $u_n = 1 + 2 + 2^2 + ... + 2^{n-1} + 2^n \times 2 = 3 \times 2^n - 1$. Ici, cela donne donc $3 \times 2^5 - 1$ soit 95 processus. Pour s'en convaincre, on peut lancer `ps | wc -l` et soustraire 4 (l'en-tête, la ligne pour `bash`, la ligne pour `wc` et la ligne pour `ps`).

--- * ---

2 Arguments de la ligne de commande

Exercice 4 : Une petite calculatrice

1. Recopier le script suivant, appelé `calcul`

```
#!/bin/sh
printf "Nombre d'arguments : $#\n"
printf '%d\n' $(($1 + $2))
```

2. Rendre ce script exécutable, puis entrer les commandes

```
$ ./calcul 13 8
$ ./calcul 6 8 12 1
$ ./calcul 89
```

3. À quoi correspondent `$1`, `$2` et `$#` ?

4. Modifier ce script, de façon à ce que s'il y a 0 ou 1 argument, le script affiche un message d'erreur. Enlever l'affichage du nombre d'arguments.
   Indication : Utiliser une construction `case` portant sur `$#`.

5. Modifier ce script, de façon à ce que, s'il y a un troisième argument, si cet argument est un des caractères `+ - * / %`, l'opération correspondante soit faite au lieu de l'addition (et si cet argument ne correspond à aucun de ces caractères, un message d'erreur soit affiché). Voir les exemples ci-dessous.
   Indications : Commencer par compléter la construction `case` de la question précédente, et utiliser une construction `case` imbriquée dans la précédente. Il faut inhiber `*`.

Exemples :

```
$ ./calcul 2
Nombre d'arguments insuffisant !
$ ./calcul 103 10
113
$ ./calcul 103 10 \*
1030
$ ./calcul 103 10 /
10
$ ./calcul 103 10 %
3
$ ./calcul 103 10 x
x : Opération non prise en charge
$ ./calcul 103 10 6 +
Trop d'arguments : entrer 2 ou 3 arguments
```

--- * ---

Correction de l'exercice 4 :

```
#!/bin/sh

case $# in
0 | 1)
    echo "Nombre d'arguments insuffisant !"
    ;;
2)
    printf '%d\n' $(($1 + $2))
    ;;
3)
```

```
    case $3 in
    + | - | \* | / | %)
        printf '%d\n' $(($1 $3 $2))
        ;;
    *)
        echo "$3 : Opération non prise en charge."
    esac
    ;;
*)
    echo "Trop d'arguments : entrer 2 ou 3 arguments"
esac
```

--- * ---

Exercice 5 : `for` sans `in`

1. Recopier le script suivant que vous appelerez `arguments`

```
#!/bin/sh
printf "Nombre total d'arguments : %d\n" $#
for chaine; do
    printf '%s\n' "$chaine"
done
```

2. Exécutez ce script des façons suivantes :

```
$ ./arguments a b c
$ ./arguments "a b c"
$ ./arguments
$ ./arguments *
$ ./arguments 367 ab 2
```

3. Ajouter un compteur de façon à afficher `arg. n° <NUM>` avant chaque argument.
4. Pour chaque argument, afficher
   - `numérique` si l'argument n'est composé que des chiffres 0, 1, ..., 9
   - `non numérique` si l'argument contient un autre caractère que ces chiffres.

--- * ---

Correction de l'exercice 5 :

```
#!/bin/sh

printf "Nombre total d'arguments : $#\n"
i=0
for chaine; do
        i=$(($i+1))
        printf '%s\n' "arg. n° $i : $arg"
        case $arg in
        *[!0-9]*) printf " : non numérique\n"
                ;;
        *) printf " : numérique\n"
                ;;
        esac
done
```

--- * ---

Exercice 6 : Script `somme`

Écrire le script somme qui affiche la somme de ses différents arguments. Quelques subtilités (voir exemples de déroulement) :

— Sans argument, 0 est affiché.

— Avec comme premier argument `--help`, un court aide-mémoire est affiché (voir les exemples ci-dessous) et le script se termine. Utiliser la commande `exit` pour mettre fin au script.

— Les arguments non numériques ne comptent pas dans la somme et un avertissement est affiché.

Exemples de déroulement :

```
$ ./somme 5 2 7
14
$ ./somme 10 56
66
$ ./somme 5 aze 7
Attention, argument aze non numérique.
12
$ ./somme
0
$ ./somme --help
Usage: somme [NUM]...
Afficher la somme des nombres entiers donnés en argument.
```

--- * ---

Correction de l'exercice 6 :

```
#!/bin/sh

case $1 in
--help)
        printf 'Usage: somme [NUM]...
Afficher la somme des nombres entiers donnés en argument.\n'
        exit
esac

somme=0
for arg; do
        case $arg in
        *[!-0-9]*)
                printf 'Attention, argument %s non numérique.\n' "$arg"
                ;;
        *)
                somme=$(( $somme + $arg ))
        esac
done
printf '%s\n' $somme
```

--- * ---

Exercice 7 : Nom de commande et révisions sur la séparation en champs

1. Recopier le script suivant dans un fichier appelé `args` :

```
#/bin/sh
printf 'Nom de commande : %s\n' "$0"
i=0
for val; do
        i=$(( $i + 1 ))
        printf 'arg. n° %d : %s\n' "$i" "$val"
done
```

2. Pour chacune des commandes suivantes, noter sur feuille l'affichage que vous prévoyez puis vérifier votre réponse en testant la commande (les parenthèses autour d'une commande servent à l'exécuter dans un sous-shell, de façon à ne pas modifier l'état, et en particulier l'`IFS` du shell courant) :

a) `$ ./args`
b) `$ ./././args`
c) `$ val="a b c d"; ./args $val`
d) `$ ./args "$val"`
e) `$ (IFS=; val="a b c d"; ./args $val)`
f) `$ (IFS='a0'; var='a102a304'; ./args $var)`
g) `$ (IFS='/'; ./args $HOME)`
h) `$ (IFS=':'; ./args $PATH)`
i) `(IFS='`
`'; ./args $(cat args))`

--- * ---

Correction de l'exercice 7 :

2.a) Il y a 0 argument et le nom de commande est `./args`
b) Il y a 0 argument et le nom de commande est `./././args`
c) Séparation en champs (par défaut, séparateurs blancs : espace, tabulation et saut de ligne), 4 arguments.
d) Pas de séparation en champs car la variable à développer est entre guillemets anglais donc 1 seul argument.
e) Pas de séparation en champs car l'IFS est vide donc 1 seul argument.
f) Séparation suivant `a` et `0`, il y a un premier argument vide (le champ avant `a`), puis `1`, puis `2`, puis `3`, puis `4`, donc 5 arguments.
g) Séparation suivant `/` : un premier argument vide (le champ avant le premier `/`) puis un argument par composant du chemin absolu vers le répertoire personnel.
h) Un argument par répertoire dans le `PATH`, la séparation en champs se fait sur le caractère `:`.
i) Un argument par ligne de `args` (la séparation en champs se fait sur les fins de ligne).

--- * ---

Exercice 8 : La commande `shift` : décaler les paramètres positionnels

Tester les commandes suivantes :

1.
```
$ set -- a b c d 'e f g' h
$ printf '$1 vaut %s\n' "$1"
$ shift
$ printf '$1 vaut %s\n' "$1"
$ shift 2
$ printf '$1 vaut %s\n' "$1"
$ printf "nb de param. pos. restants : $#\n"
$ echo "param. pos. restants :"; printf '%s\n' "$@"
```

2. On considère la variable chaine contenant une phrase, par exemple

```
$ chaine="Alice et Bob travaillent sous Unix."
```

Comment, à l'aide (notamment) d'une séparation en champs, de `set` et `shift` afficher le dernier mot de la chaîne ?

--- * ---

Correction de l'exercice 8 :

1. La première commande `set` définit 6 paramètres positionnels, dont le premier qui vaut `a` est affiché. Ensuite la commande `shift` sans argument décale tous ces paramètres positionnels : l'ancien `$1` est oublié, l'ancien `$2` devient le nouveau `$1`, l'ancien `$3` devient le nouveau `$2`, ... Le paramètre `$1` est donc maintenant développé en `b`.

Puis la commande `shift 2` décale la liste des paramètres positionnels de 2 donc `$1` sera maintenant développé en `d`. Plus précisément la liste des paramètres positionnels est maintenant `d 'e f g' h`. En particulier, `$1` vaut `d`, `$#` vaut 3 et le dernier affichage produit les trois lignes

```
d
e f g
h
```

Noter que `"$@"` (**avec les guillemets anglais**) a bien été développé en 3 mots et que `e f g` n'a pas subi la séparation en champs.

2. C'est tordu... (on peut faire un peu plus simple avec la commande `eval` qu'on verra plus tard).

```
$ chaine="Alice et Bob travaillent sous Unix."
$ set -- $chaine; shift $(($# - 1)); printf "$1\n"
Unix.
```

Explication : dans la commande `set` on utilise la séparation en champs pour avoir autant de paramètres positionnels qu'il y a de mots dans la phrase. La commande `shift` utilise un développement arithmétique pour décaler ces paramètres positionnels du nombre de mots de la phrase -1. En conséquence, `$1` est maintenant le dernier mot de la phrase.

--- * ---

Exercice 9 : `set --` : redéfinir les paramètres positionnels

La commande `set` suivie de l'option `--` [1] permet de (re-)définir la valeur (et le nombre) des paramètre postitionnels.

1. Tester les commandes suivantes :

```
$ set -- a b c 'a b c'
$ printf '%d\n' $#
$ printf '$1 vaut %s\n' "$1"
$ printf '$2 vaut %s\n' "$2"
$ printf '$3 vaut %s\n' "$3"
$ printf '$4 vaut %s\n' "$4"
```

2. Tester les commandes suivantes (le paramètre spécial `$@` est développé en la suite des paramètres positionnels) :

```
$ set -- *
$ printf "nbre de fichiers du rep. courant : $#\n"
$ printf "%s\n" "$@"
$ set -- */
$ printf "nbre de sous-répertoires du rep. courant : $#\n"
$ printf "%s\n" "$@"
```

1. En fait l'option `--` met fin aux options.

3. À l'aide de la commande `date`, d'une substitution de commande, et de `set --`, afficher l'heure qu'il est (et seulement cette information) dans le terminal.

--- * ---

Correction de l'exercice 9 :

1. La commande `set --` a redéfini les paramètres positionnels, il y en a maintenant 4, qui sont (dans l'ordre) `a`, `b`, `c` et `a b c` (chaîne contenant deux espaces).
2. La première commande `set --` a redéfini les paramètres positionnels, il y en a maitenant autant que de fichiers (non cachés) dans le répertoire courant. Le premier `printf` affiche le nombre de ces fichiers et le suivant les affiche tous chacun sur une ligne.

   Le second `set --` définit autant de paramètres positionnels qu'il y a de sous-répertoires (les fichiers dont le nom se termine par une oblique ne peuvent être que des répertoires). Leur nombre est ensuite affiché, puis chacun de ces sous-répertoires sur une ligne séparée.
3. `set -- $(date); printf "$4\n"`
   Explication : dans la première commande, le shell fait une substitution de commande puis, comme celle-ci n'est pas entre guillemets anglais, une séparation en champs. Nous n'avons pas modifié `IFS` donc les séparateurs sont les blancs (espace, tabulation, fin de ligne). La commande `set` définit donc autant de paramètres positionnels qu'il y a de champs : ici 6 et chez moi le $4^{\text{ème}}$ (mais cela dépend de votre programme `date`) est l'heure.

--- * ---

3 Exercices d'entraînement sur les scripts

Exercice 10 : `suffixer`

Écrire le script `suffixer` qui prend au moins 2 arguments : le premier est le suffixe et les suivants sont les chaînes auxquelles on veut ajouter le suffixe. Le script produit l'affichage des chaînes avec leur suffixe, ou un message d'erreur si le nombre d'arguments n'est pas au moins deux. Exemples :

```
$ ./suffixer .c prog hello 'a b'
prog.c
hello.c
a b.c
$ ./suffixer ' est heureux(se)' Alice Bob
Alice est heureux(se)
Bob est heureux(se)
$ ./suffixer .c
Usage: suffixer suffixe chaine...
```

--- * ---

Correction de l'exercice 10 :

```
#!/bin/sh
case $# in
        0 | 1) echo "Usage: suffixer suffixe chaine..."
               exit 1
esac

suffixe=$1
shift
```

```
for mot; do
        printf '%s%s\n' "$mot" "$suffixe"
done
```

--- * ---

Exercice 11 : `chgext`

Écrire le script `chgext` qui prend en premier argument une nouvelle extension pour des fichiers (par exemple `txt` ou `c` ou `jpg`, ...) et des noms de fichiers existants.

Chacun de ces fichiers est renommé en remplaçant son extension éventuelle par la nouvelle. Un message d'erreur est affiché si le nombre d'arguments est insuffisant.

Exemples :

```
$ ls
a.txt b.truc notes
$ chgext c *
$ ls
a.c b.c notes.c
$ chgext h b.c; ls
a.c b.h notes.c
$ chgext jpeg
Usage: chgext nouv_ext nom_fic...
```

Indication : `IFS=.` et ne pas oublier que la commande `break` permet de sortir d'une boucle.

--- * ---

Correction de l'exercice 11 :

```
#!/bin/sh

case $# in
        0 | 1)
                echo "Usage: chgext nouv_ext nom_fic..."
                echo "Exemple: chgext cc prog.cpp truc.cpp"
                exit 1
esac

ext=$1
shift
IFS='.'
for mot; do
        for base in $mot; do
                nouv="$(printf '%s.%s' "$base" "$ext")"
                mv "$mot" "$nouv"
                break
        done
done
```

--- * ---

Exercice 12 : Nom de base

Le *nom de base* d'un nom de chemin est celui que l'on obtient en enlevant le nom d'un éventuel répertoire parent (bref tout ce qui vient avant la dernière oblique).

Par exemple :

— Le nom de base de `/usr/bin` est `bin`.

— Le nom de base de `/home/alice/prog.c` est `prog.c`.

— Le nom de base de `../prog.c` est `prog.c`.

1. Écrire un script `nom_de_base` qui, pour chacun de ses arguments, écrit sur le terminal le nom de base correspondant, c'est-à-dire cet argument privé de son plus long préfixe se terminant par `/`.
   Indication : on pourra se servir de la variable `IFS` ainsi que d'une boucle `for` dont le corps ne contient que l'instruction vide `:` (deux-points).
2. **Bonus.** Problème : avec le script précédent (et la définition de nom de base que nous avons donnée), le nom de base de `/usr/bin/` (avec une oblique à la fin, comme c'est autorisé pour les noms de répertoire) est la chaîne vide, alors que nous voudrions que ce soit `bin`.
   Autrement dit, il faudrait ignorer une (ou plusieurs) obliques éventuelles à la fin du nom de chemin.
   Régler ce problème.
   Indications : On peut tester qu'une chaîne est vide en la comparant à l'aide de `case` avec la chaîne vide `""` (ou `''`). On pourra aussi mémoriser la chaîne précédente de la liste dans une autre variable.

--- * ---

Correction de l'exercice 12 :

```
#!/bin/sh
IFS='/'
for arg; do
        for mot in $arg; do
                # Ceci traite le cas d'une chaine qui se termine par un ou plusieurs /
                case $mot in
                '') mot=$precedent ;;
                *) precedent=$mot
                esac
                # Si on ne traite pas ce cas il faudrait juste mettre la commande vide
                # :
        done
        printf "$mot\n"
done
```

--- * ---