Initiation à l'environnement Unix : TP8

novembre 2025 — Pierre Rousselin

Exercice 1 : *job control*

1. Lancer les commandes
   ```
   $ sleep 2000 &sleep 3000& sleep 5000& sleep 6000& sleep 7000& sleep 8000&
   $ jobs
   ```
   Combien de processus sont en arrière-plan ? Quels sont leurs numéros de *job* ?
2. À l'aide de la commande `fg`, mettre en avant-plan le processus de commande `sleep 3000` : voir son numéro de job avec *jobs* et utiliser la commande `fg %num` où `num` est son numéro de *job*.
3. Stopper ce processus (Ctrl-Z) et voir la sortie de `jobs`.
4. À l'aide de la commande `bg`, réveiller le processus de commande `sleep 3000`.
5. À l'aide de la commande `kill`, tuer le processus de commande `sleep 6000` en utilisant un numéro de *job*.
6. Voir les PID des processus en arrière-plan avec `jobs -l` et tuer le processus de commande `sleep 7000` en utilisant son PID.
7. Lancer la commande `kill -STOP <pid>` en remplaçant `<pid>` par le PID de la commande `sleep 5000`, puis relancer `jobs -l` pour voir ce qui a changé. Voir l'état des processus du terminal courant avec `ps -lf`.
8. Lancer la commande `kill -CONT <pid>`, où `<pid>` est le PID de la commande `sleep 5000` et voir les changements avec `jobs` et `ps`.
9. Il y a toujours un *job* dont le numéro (entre crochets) est suivi par un `+`, en gros le dernier *job* a avoir subi une opération de *job control*. Lancer la commande `kill %+` et voir le résultat avec `jobs` et `ps`. Quel est le nouveau *job* désigné comme `+` ?
10. Écrire une boucle `while` qui, tant qu'il y a des processus en arrière-plan, les tue, en attendant une seconde entre chaque itération.
11. Relancer la première commande de cet exercice, voir la sortie de `jobs -p` et écrire une commande `kill` pour tuer tous les processus en arrière-plan.

--- * ---

Correction de l'exercice 1 :

1. La commande lance 6 processus en arrière-plan. TODO : les lancer avec une boucle ne permet pas de les identifier ensuite avec *jobs* (`bash` affiche les commandes `sleep ${i}000`). Est-il possible de faire mieux ?
2. Probablement `fg %2`, mais cela dépend des tâches en arrière-plan.
3. `sleep 3000` est maintenant stoppé (en pause).
4. `bg %2` relance ce processus en arrière-plan.
5. `kill %4` (ou autre chose s'il y a eu d'autres tâches en arrière plan) lance le signal `TERM` à ce processus, ce qui (par défaut) le tue.
6. `kill $(jobs -l | grep 7000 | awk '{ print $2 }')`
7. Le processus `sleep 5000` est stoppé (comme s'il avait été en avant-plan et qu'on avait tapé CTRL-Z).
8. Le processus `sleep 5000` est relancé en arrière-plan (comme avec `bg`).
9.
10. `while kill %+; do sleep 1; done` On peut faire mieux avec `wait`, mais peut-être pas dès la L1.
11. `kill $(jobs -p)`

--- * ---

Exercice 2 : Supprimer des préfixes et des suffixes

Essayer de prévoir la sortie des commandes suivantes et les essayer.

1. `$ verbe=scripter; printf '%s\n' "${verbe%er}"`
2. `$ verbe=scripter; printf '%s\n' "${verbe%??}"`
3. `$ verbe=scripter; printf '%s\n' "Vous ${verbe%??}ez"`
4. `$ verbe=manger; printf '%s\n' "r${verbe#?}"`
5. `$ verbe=scripter; printf '%s\n' "${verbe#${verbe%??}}"`

--- * ---

Exercice 3 : Conjuguer

1. Écrire le script conjuguer qui prend un argument qui doit être un verbe du premier groupe (un verbe qui se termine par « er ») et écrit sur sa sortie la conjugaison au présent de l'indicatif de ce verbe. Pour l'instant, ne considérer que les cas les plus simples, et supposer qu'il n'y a pas d'erreur.
2. Faire en sorte, avec un développement modifié, que sans argument, le verbe « par défaut » soit « scripter ».
3. Faire en sorte, avec une construction `case`, d'arrêter le script avec un message d'erreur, si le verbe n'est pas du premier groupe.
4. Gérer, de préférence avec concision, le cas où le radical (la partie du verbe sans « er ») se termine par la lettre g.

--- * ---

Correction de l'exercice 3 :

```
#!/bin/sh

verbe=${1:-scripter}

case $verbe in
*er) :;;
*)
        printf '%s: l'\''argument %s fourni n'\''est pas du premier groupe' \
                "$0" "$1" >&2
        exit 1
esac

case $verbe in
*ger) term_nous=eons;;
*) term_nous=ons
esac

rad=${verbe%er}
printf '%s\n' "Je ${rad}e"
printf '%s\n' "Tu ${rad}es"
printf '%s\n' "Il ${rad}e"
printf '%s\n' "Nous ${rad}${term_nous}"
printf '%s\n' "Vous ${rad}ez"
printf '%s\n' "Ils ${rad}ent"
```

--- * ---

Exercice 4 : Palindromes

1. Quel développement modifié permet d'obtenir :

a) le contenu d'une variable sans sa première lettre ?

b) la première lettre du contenu d'une variable ?

c) le contenu d'une variable sans sa dernière lettre ?

d) la dernière lettre du contenu d'une variable ?

2. Un palindrome est un mot qui est identique si on le lit à l'envers, comme « kayak » ou « alla ». Le mot vide et le mots d'une seule lettre sont aussi des palindromes. Écrire un script `palindrome` qui lit (pour l'instant) une ligne d'entrée et la réécrit sur sa sortie si c'est un palindrome (sinon n'affiche rien du tout).

3. Changer le script pour qu'il fasse de même sur toutes les lignes de son entrée standard (tant qu'il y en a).

4. Si la commande `aspell` est installée avec un dictionnaire français [1], tester la commande

```
$ aspell -d fr dump master
```

puis chercher tous les mots qui sont des palindromes en français. Quels sont les plus longs ?

5. Voir avec la commande `time` la durée d'exécution de la commande précédente et la noter.

--- * ---

Correction de l'exercice 4 :

1. a) `${parameter#?}`

b) `${parameter%{$parameter#?}}`

c) `${parameter%?}`

d) `${parameter#{$parameter%?}}`

2.

3.
```
#!/bin/sh
while IFS= read -r ligne; do
        mot=$ligne
        est_palin=1
        while [ ${#mot} -gt 1 ]; do
                prem=${mot%${mot#?}}
                der=${mot#${mot%?}}
                mot=${mot%$prem}
                mot=${mot#$der}
                if [ $prem != $der ]; then
                        est_palin=0
                        break
                fi
        done
        if [ $est_palin = 1 ]; then printf '%s\n' "$ligne"; fi
done
```

4. `$ aspell -d fr dump master | ./palindrome >tous_palins`

5. `$ awk '{printf("%d %s\n", length($0), $0)}' tous_palins | sort -nr` On voit qu'il y a 2 palindromes qui font 9 caractères (« ressasser » et « essayasse »).

--- * ---

1. S'il n'est pas installé, essayer d'installer le paquet `aspell-fr`.

Exercice 5 : Afficher les lignes communes de deux fichiers

La commande suivante permet d'ouvrir deux fichiers en lecture, avec les descripteurs 3 et 4 :

```
exec 3<fichier1 4<fichier2
```

On peut ensuite lire dans chaque fichier avec `read ligne <&3` et `read ligne <&4`.

Écrire le script `same_lines` qui échoue s'il n'a pas deux arguments qui sont des fichiers normaux qu'on peut lire, puis :

- si la première ligne du premier fichier est identique à la première ligne du second fichier l'affiche (sinon n'affiche rien) ;
- si la deuxième ligne du premier fichier est identique à la deuxième ligne du second fichier, l'affiche (sinon n'affiche rien) ;
- etc.
- s'arrête dès qu'on a fini de lire un des deux fichiers.

--- * ---

Correction de l'exercice 5 :

```
#!/bin/sh

case $# in
2) :;;
*)
        printf 'Usage : %s FICHIER1 FICHIER2\n' >&2
        exit 1
esac

if ! [ -f "$1" ]; then
        printf '%s: %s: fichier normal introuvable\n' "$0" "$1" >&2
        exit 1
fi

if ! [ -r "$1" ]; then
        printf '%s: %s: permissions insuffisantes\n' "$0" "$1" >&2
        exit 1
fi

if ! [ -f "$2" ]; then
        printf '%s: %s: fichier normal introuvable\n' "$0" "$2" >&2
        exit 1
fi

if ! [ -r "$2" ]; then
        printf '%s: %s: permissions insuffisantes\n' "$0" "$2" >&2
        exit 1
fi

fichier1=$1
fichier2=$2

exec 3<"$fichier1" 4<"$fichier2"
```

```
while IFS= read -r ligne1 <&3 && IFS= read -r ligne2 <&4; do
        if [ "$ligne1" = "$ligne2" ]; then
                printf '%s\n' "$ligne1"
        fi
done
```

--- * ---

Exercice 6 : Palindromes avec `rev` et `same_lines`

1. Si la commande `rev` est installée, tester la commande
   `$ aspell -d fr dump master | rev`
2. À l'aide de redirections dans des fichiers, de `rev` et du script de l'exercice précédent, donner une autre façon d'obtenir tous les palindromes de langue française.
3. Est-ce plus rapide que la solution précédente ?
4. Écrire une nouvelle version du script `palindromes` qui utilise `rev` à la place des développements modifiés. Que dire de ses performances et pourquoi, d'après vous ?
5. Si vous savez ouvrir des fichiers et faire des entrées-sorties en C, essayer d'écrire `same_lines` en C, par exemple en utilisant les fonctions `fopen`, `fclose`, `fgets` et `strcmp`. Comparer les durées d'exécution.

--- * ---

Correction de l'exercice 6 :

1.
2. ```
   $ aspell -d fr dump master >mots
   $ rev <mots >mots_rev
   $ ./same_lines mots mots_rev >tous_palins
   ```
3. Sur ma machine, c'est à peine plus rapide que la solution précédente.
4. ```
   #!/bin/sh

   while IFS= read -r ligne; do
           if [ $(printf '%s\n' "$ligne" | rev) = "$ligne" ]; then
                   printf '%s\n' "$ligne"
           fi
   done
   ```
   C'est beaucoup trop lent sur un fichier de 700000 lignes car il faut chaque fois créer un processus qui exécute `rev`.
5. ```
   #include <stdio.h>
   #include <string.h>

   #define MAXLINE 1024
   int main(int argc, char *argv[])
   {
           if (argc != 3) {
                   fprintf(stderr, "Usage: %s fichier1 fichier2\n", argv[0]);
                   return 1;
           }
           FILE *f = fopen(argv[1], "r");
           if (f == NULL) {
                   perror("fopen");
                   return 1;
   ```
   ```

```
        }
        FILE *g = fopen(argv[2], "r");
        if (f == NULL) {
                perror("fopen");
                return 1;
        }
        char ligne1[MAXLINE], ligne2[MAXLINE];
        while (fgets(ligne1, MAXLINE, f) && fgets(ligne2, MAXLINE, g)) {
                if (strcmp(ligne1, ligne2) == 0)
                        printf("%s", ligne1);
        }
        return 0;
}
```

En remplaçant le script `same_lines` par ce programme en C, par contre, c'est incomparablement plus rapide.

--- * ---