Initiation à l'environnement Unix
CM6 : Processus (1) introduction et arguments de la ligne de commande

Pierre Rousselin

Université Paris 13
L1 informatique
novembre 2021

Scripts shell : construction `for`

Processus : (non-)définition

Les paramètres positionnels

Manipuler les paramètres positionnels

Exécution d'une commande *en arrière-plan*

Job control

Premier exemple

```
#!/bin/sh
# script 'liste'
for f in 'fich. normaux :' * 'fich. cachés :' .*; do
    printf "$f\n"
done
```

```
$ ./liste
fich. normaux :
a.out
hello.c
fich. cachés :
.
..
.bashrc
```

Syntaxe

```
for var in mot1 mot2 ... motn
do
    commande1
    commande2
    ...
done
```

ou

```
for var in mot1 mot2 ... motn; do
    commande1
    commande2
    ...
done
```

Remarque : le mot-clé `do` *doit* suivre soit un caractère de fin de ligne, soit un point-virgule.

Sémantique

- Chacun des mots compris entre `in` et la fin de ligne suivante (ou le point-virgule suivant) subit tous les traitements usuels faits par le shell (développement de variables, **séparation en champs**, développement de noms de chemins, ...).
- Remarque : le nombre de mots peut ainsi augmenter.
- La suite de commande comprise entre les mots-clés `do` et `done` est ensuite répétée autant de fois qu'il y a de mots dans la liste,
- la variable dont le nom est entre `for` et `in`, contenant successivement et dans l'ordre chacun des mots de la liste.

Sémantique

- Chacun des mots compris entre `in` et la fin de ligne suivante (ou le point-virgule suivant) subit tous les traitements usuels faits par le shell (développement de variables, **séparation en champs**, développement de noms de chemins, ...).
- Remarque : le nombre de mots peut ainsi augmenter.
- La suite de commande comprise entre les mots-clés `do` et `done` est ensuite répétée autant de fois qu'il y a de mots dans la liste,
- la variable dont le nom est entre `for` et `in`, contenant successivement et dans l'ordre chacun des mots de la liste.
- La commande `break` permet de sortir de la boucle.
- La commande `continue` permet de passer directement à l'itération suivante.

Processus : *être*

Définition la plus courante (par exemple Maurice J. Bach, “The design of the Unix operating system”) :

> *Un programme est un fichier exécutable et un processus est une instance de ce programme en cours d'exécution.*

- Différence entre le *programme*, **inerte** et le *processus*, exécution de ce programme, **dynamique**.
- Instance : anglicisme dur à traduire, disons « cas particulier avec un état initial ou des paramètres donnés »

Processus : analogie

- Analogie courante : programme $\leftrightarrow$ recette de cuisine (par exemple écrite dans un livre). Exemple :
  - faire fondre le chocolat ;
  - battre les blancs en neige ;
  - mélanger les jaunes avec le chocolat ;
  - incorporer les blancs dans le mélange ;
  - laisser reposer 10h au réfrigérateur.

  La recette n'est qu'une suite d'instructions, elle dit comment faire à manger, mais on ne peut pas la manger...

Processus : analogie

- Analogie courante : programme $\leftrightarrow$ recette de cuisine (par exemple écrite dans un livre). Exemple :
  - faire fondre le chocolat ;
  - battre les blancs en neige ;
  - mélanger les jaunes avec le chocolat ;
  - incorporer les blancs dans le mélange ;
  - laisser reposer 10h au réfrigérateur.

  La recette n'est qu'une suite d'instructions, elle dit comment faire à manger, mais on ne peut pas la manger...
- en poursuivant cette analogie, le processus correspond à la *la recette en train d'être suivie en cuisine* : un cuisinier ou une cuisinière, voire plusieurs s'entraidant, quelque part dans le monde :
  - font vraiment fondre du vrai chocolat ;
  - battent de vrais blancs de vrais œufs en neige ;
  - ...

  Le fruit de ce travail peut être mangé (et la mousse au chocolat, c'est délicieux !)

Processus : analogie (2)

Poursuivant encore cette analogie,

- il peut y avoir plusieur processus de « faire une mousse au chocolat » dans le monde, suivant la même recette (le même programme peut être utilisé par plusieurs processus) ;
- chaque cuisinier faisant une mousse au chocolat pourra ou devra noter quelque part :
  - pour combien de personnes ? (un argument)
  - à quel moment il a commencé ? (date de départ)
  - qui lui a demandé de faire cette mousse ? (UID, identifiant de l'utilisateur à qui appartient ce processus)
  - dans quelle cuisine il fait cette mousse (répertoire courant)
  - ...

Processus : analogie (3)

- Si le cuisinier est interrompu (par exemple par un enfant en bas âge), necessité de noter à quel endroit on en était dans la recette.
- Si le cuisinier travaille avec d'autres cuisiniers, ils se partagent des ressources : « tu me dis quand tu as fini avec le batteur électrique ? »
- le processus est alors en attente, ce serait bien que le cuisinier ne reste pas à glandouiller, mais par exemple, commence une autre recette ;
- encore plus vrai pendant la dernière étape (laisser reposer 10h).

Processus Unix

- Idée de départ : plusieurs utilisateurs peuvent utiliser *simultanément* une seule machine ;
- Nécessaire que le système soit multi-processus :
  - Ken édite (avec `ed`) un fichier sur un terminal pendant que
  - Dennis compile un fichier source sur un autre terminal et
  - Brian copie des fichiers sur un troisième.
- L'ordonnanceur (*scheduler*) du système partage le temps de calcul du ou des processeurs entre les processus. Grossièrement :
  - les processus sont placés dans une file d'attente ;
  - celui qui est en tête est *élu* : utilise le processeur pendant une courte période ;
  - s'il n'est pas terminé, retourne au bout de la file.
  - Algorithme *round-robin* ou *tourniquet*.
- Gestion des processus en attente d'événement : les endormir et les réveiller le moment venus.

Processus Unix

- Les informations correspondant aux processus sont rangés dans un tableau. L'indice du processus dans ce tableau est un petit entier appelé `PID` (*Process identifier*).
- Les permissions d'un processus sont celles de son propriétaire utilisateur et de son groupe propriétaire.
- Un processus a un état, les plus courants :
  - `S` : pour *sleeping*, en attente d'un événement (entrée utilisateur, fin d'un autre processus, ...)
  - `R` : pour *running*, les instructions du programme sont exécutées par un ou plusieurs processeurs, ou bien le processus est dans la file d'attente
  - `T` : interrompu (mis en pause) par l'utilisateur.
- Un processus a également une date de départ (`STIME`), un terminal de contrôle (`TTY`), un temps total de calcul (`TIME`)...

Arborescence des processus

- Seul un processus peut créer un nouveau processus.
- À part, évidemment, le tout premier processus, celui de `PID` 1 qui est créé au démarrage du système :
  - `init` sur les Unix traditionnels ;
  - `systemd` sur la plupart des systèmes GNU/Linux aujourd'hui ;
  - `launchd` sur MacOS X.
- Lorsqu'un processus demande au noyau la création d'un nouveau processus et que celle-ci est acceptée, on dit qu'il *fork*.
  - Ce nouveau processus *naît* et le processus qui l'a créé est son *parent*.
  - Le nouveau processus est *l'enfant* de celui qui l'a créé.
- Ainsi, les processus créent une *arborescence* dont la racine est le processus de `PID 1`, qui est l'ancêtre commun de tous les processus.
- Le `PID` du processus parent s'appelle le `PPID` (*parent process id*).

Ce que fait le shell avec une commande externe

- Lorsque le shell a fini sa cuisine, et que la commande est une commande externe, il *fork* : crée un nouveau processus ;
- puis il se met en sommeil ;
- et se réveille quand son processus enfant *meurt*, c'est-à-dire se termine.

La commande `ps`

- La commande `ps` permet d'afficher des informations sur les processus en cours.
- Sans argument, n'affiche que quelques informations sur les processus dont le terminal de contrôle est celui dans lequel la commande est lancée.

```
$ ps
    PID TTY          TIME CMD
   5318 pts/1    00:00:00 bash
  16269 pts/1    00:00:00 ps
```

- Plus d'informations : option `-f` pour *full-format*

```
$ ps -f
UID          PID    PPID  C STIME TTY          TIME CMD
pierre      5318    2593  0 10:08 pts/1    00:00:00 bash
pierre     17181    5318  0 12:22 pts/1    00:00:00 ps -f
```

- Qui est le parent du processus `ps` ici ?

La commande `ps`

Sélection des processus :

- option `-e` (*every*) pour avoir des informations sur tous les processus ;
- option `-u alice` pour avoir des informations sur tous les processus dont l'utilisateur propriétaire est `alice` ;
- option `-p 1234` pour avoir des informations sur le processus dont le `PID` est `1234` ;
- option `-C cmd` pour avoir des informations sur tous les processus dont le *nom de commande* est `cmd`.

La commande `ps`

Contrôle de l'affichage :

- option `-f` (*full-format*) : plus de colonnes, les arguments de la commande sont aussi affichés ;
- option `-l` (*long*) : encore plus de colonnes ;
- option `-o champ[,champ]...` pour un contrôle fin de l'affichage avec seulement les colonnes spécifiées :

`pid`	pour le `PID`
`ppid`	pour le `PPID`
`user`	pour l'utilisateur propriétaire
`stime`	pour la date de début
`time`	pour le temps total d'utilisation du processeur
`comm`	pour le nom de commande
`cmd`	pour la commande entière (nom + arguments)

La commande `ps`

Contrôle de l'affichage :

- option `-f` (*full-format*) : plus de colonnes, les arguments de la commande sont aussi affichés ;
- option `-l` (*long*) : encore plus de colonnes ;
- option `-o champ[,champ]...` pour un contrôle fin de l'affichage avec seulement les colonnes spécifiées :

`pid`	pour le `PID`
`ppid`	pour le `PPID`
`user`	pour l'utilisateur propriétaire
`stime`	pour la date de début
`time`	pour le temps total d'utilisation du processeur
`comm`	pour le nom de commande
`cmd`	pour la commande entière (nom + arguments)

- encore bien d'autres choses... voir le manuel de `ps` mais...

La commande `ps`

Contrôle de l'affichage :

- option `-f` (*full-format*) : plus de colonnes, les arguments de la commande sont aussi affichés ;
- option `-l` (*long*) : encore plus de colonnes ;
- option `-o champ[,champ]...` pour un contrôle fin de l'affichage avec seulement les colonnes spécifiées :

`pid`	pour le `PID`
`ppid`	pour le `PPID`
`user`	pour l'utilisateur propriétaire
`stime`	pour la date de début
`time`	pour le temps total d'utilisation du processeur
`comm`	pour le nom de commande
`cmd`	pour la commande entière (nom + arguments)

- encore bien d'autres choses... voir le manuel de `ps` mais...

```
$ man ps | wc -l
1269
```

`PID` et `PPID` en shell

- En shell, le `PID` du processus courant (shell ou script) s'obtient comme développement du paramètre spécial `$$` ;
- le `PPID` est obtenu, sous `bash` (et dans presque tous les shell) avec la variable en lecture seule `$PPID`.

PID et PPID en shell

- En shell, le PID du processus courant (shell ou script) s'obtient comme développement du paramètre spécial $$;
- le PPID est obtenu, sous bash (et dans presque tous les shell) avec la variable en lecture seule $PPID.

Dans cet exemple, le shell crée un processus enfant qui exécute bash.

```
$ echo $$ $PPID
48182 32257
$ bash
$ echo $$ $PPID
52245 48182
```

Nom de commande et arguments

- Un processus a toujours un `nom de commande` (une chaîne de caractère) :
  - en shell : `$0`
  - en C : `argv[0]`
- Et peut avoir un certain nombre `d'arguments`, des chaînes de caractère, fournies au noyau lorsqu'on lui demande d'exécuter un programme :
  - en shell : `$1`, `$2`, ...
  - en C : `argv[1]`, `argv[2]`, ...
- Le nombre de ces arguments est :
  - en shell : `$#`
  - en C : `argc - 1`.

Scripts shell : construction `for`

Processus : (non-)définition

Les paramètres positionnels

Manipuler les paramètres positionnels

Exécution d'une commande *en arrière-plan*

Job control

Premier exemple : `mult`

```
#!/bin/sh
# mult version 1
printf "%d\n" $(($1 * $2))
```

Exemples :

```
$ chmod u+x mult
$ ./mult 3 56
168
```

Terminologie : `$1` et `$2` sont des *paramètres positionnels*.

Premier exemple : `mult`

```
#!/bin/sh
# mult version 1
printf "%d\n" $(($1 * $2))
```

Exemples :

```
$ chmod u+x mult
$ ./mult 3 56
168
```

Terminologie : `$1` et `$2` sont des *paramètres positionnels*.
Problème :

```
$ ./mult 3
./mult: ligne 3: 3 *  : erreur de syntaxe :
    opérande attendu (le symbole erroné est « *  »)
$ gné ?
bash: gné: commande inconnue...
```

Nombre d'arguments

```
#!/bin/sh
# mult version 2
case $# in
    2) : ;;
    *)
        echo "Nombre d'arguments incorrects"
        echo "Usage: mult nb1 nb2"
        exit 1
esac
printf "%d\n" $(($1 * $2))
```

Nombre d'arguments

```
#!/bin/sh
# mult version 2
case $# in
    2) : ;;
    *)
        echo "Nombre d'arguments incorrects"
        echo "Usage: mult nb1 nb2"
        exit 1
esac
printf "%d\n" $(($1 * $2))
```

- Le *paramètre spécial* `$#` est développé en le nombre d'arguments.
- La commande `:` (deux points) est la *commande vide*, elle ne fait rien (permet simplement de passer à la suite).
- La commande `exit` (éventuellement suivie d'un petit entier) permet de mettre fin au script.

Nombre d'arguments

```
#!/bin/sh
# mult version 2
case $# in
    2) : ;;
    *)
        echo "Nombre d'arguments incorrects"
        echo "Usage: mult nb1 nb2"
        exit 1
esac
printf "%d\n" $(($1 * $2))
```

- Le *paramètre spécial* `$#` est développé en le nombre d'arguments.
- La commande `:` (deux points) est la *commande vide*, elle ne fait rien (permet simplement de passer à la suite).
- La commande `exit` (éventuellement suivie d'un petit entier) permet de mettre fin au script.
- Convention : si aucune erreur `exit 0`
  en cas d'erreur `exit 1` ou `exit 2` ou ... ou `exit 126`.

for sans in

```
#!/bin/sh
# mult : version 3

prod=1 # élément neutre de la multiplication

for num; do
        prod=$((prod * num))
done
printf "%d\n" $prod
```

for sans in

```
#!/bin/sh
# mult : version 3

prod=1 # élément neutre de la multiplication

for num; do
        prod=$((prod * num))
done
printf "%d\n" $prod
```

Exemples :

```
$ ./mult 2 3 4 5
120
$ ./mult 3
3
$ ./mult
1
```

Améliorer `mult`

On peut encore améliorer le script précédent (exercice !)

- Tester chacun des argument pour vérifier qu'il est bien numérique.
- Si l'un des arguments vaut 0, afficher immédiatement 0 sans poursuivre le calcul.
- Ajouter les options `-x` (respectivement `-o`) pour que le résultat soit affiché en hexadécimal (respectivement en octal).

Conclusion

- Les paramètres positionnels sont `$1`, `$2`, ...
- Lorsqu'ils sont développés ils correspondent aux arguments de la ligne de commande.
- Le paramètre spécial `$#` correspond au nombre de ces arguments.
- La boucle `for` sans `in` permet d'itérer sur la liste des arguments

```
for var
do
    # commandes
done
```

ou

```
for var; do
    # commandes
done
```

Scripts shell : construction `for`

Processus : (non-)définition

Les paramètres positionnels

Manipuler les paramètres positionnels

Exécution d'une commande *en arrière-plan*

Job control

La commande `set`

La commande `set` a deux usages bien distincts, ce qui est un de trop, car d'après la philosophie Unix :

> *Un programme (ou une commande) ne devrait faire qu'une seule chose mais la faire bien.*

Premier usage : activer ou désactiver les options du shell, par exemple

- `set -f` : désactive le développement de noms de chemins
- `set +f` : réactive le développement de noms de chemins
- `set -x` : active la trace, le shell affiche la commande après tous ses développements avant d'exécuter une commande
- `set +x` : désactive la trace

L'option `--` signale la fin des options et donc le début de l'autre utilisation de la commande `set`...

La commande `set --`

Les arguments qui ne sont pas des options (ou plus simplement qui suivent l'option `--`) deviennent les nouvelles valeurs des paramètres positionnels.

```
$ set -- Ainsi font, font, font
$ printf "Nombre de paramètres positionnels : $#"
Nombre de paramètres positionnels : 4
$ printf "$1 $2 $2 $2 $2 $2 $4\n"
Ainsi font, font, font, font, font, font
```

La commande `set --`

Les arguments qui ne sont pas des options (ou plus simplement qui suivent l'option `--`) deviennent les nouvelles valeurs des paramètres positionnels.

```
$ set -- Ainsi font, font, font
$ printf "Nombre de paramètres positionnels : $#"
Nombre de paramètres positionnels : 4
$ printf "$1 $2 $2 $2 $2 $2 $4\n"
Ainsi font, font, font, font, font, font
```

Pourquoi utiliser `--` systématiquement ?

```
$ ma_var=-f
$ set $ma_var # Est-ce vraiment ce que je voulais ?
```

La commande `shift`

La commande `shift` décale les paramètres positionnels et diminue leur nombre.

```
$ set -- a1 b2 c3 d4; echo $#
4
$ printf "$1 $2 $3 $4\n"
a1 b2 c3 d4
$ shift; echo $#
3
$ printf "$1 $2 $3 $4\n"
b2 c3 d4
$ shift 2; echo $#
1
$ printf "$1 $2 $3 $4\n"
d4
```

Le paramètre spécial $@

Le paramètre spécial $@ est étendu en l'ensemble des paramètres positionnels.

```
$ set -- a b c
$ printf "%s\n" $@
a
b
c
$ set -- 'a b' c
$ printf "%s\n" $@
a
b
c
$ printf "%s\n" "$@"
a b
c
```

Le paramètre spécial $@

Le paramètre spécial $@ est étendu en l'ensemble des paramètres positionnels.

```
$ set -- a b c
$ printf "%s\n" $@
a
b
c
$ set -- 'a b' c
$ printf "%s\n" $@
a
b
c
$ printf "%s\n" "$@"
a b
c
```

Lorsqu'il est entre "", il n'y a pas de séparation en champs, et c'est presque toujours ce qu'on veut. Donc on retient plutôt "$@".

Scripts shell : construction `for`

Processus : (non-)définition

Les paramètres positionnels

Manipuler les paramètres positionnels

Exécution d'une commande *en arrière-plan*

Job control

Attente de la fin d'une commande

Dans l'exemple suivant, le système attend 5 secondes, puis la commandes `sleep 5` se termine avec le code de retour 0 (succès), ce qui entraîne l'impression de `prêt !` sur la sortie standard.
Une fois la « liste-ET » terminée, le shell peut lancer la commande suivante, affichant `voilà !` sur la sortie standard.

```
$ sleep 5 && echo 'prêt !'; echo 'voilà !'
prêt !
voilà !
```

Attente de la fin d'une commande

Dans l'exemple suivant, le système attend 5 secondes, puis la commandes `sleep 5` se termine avec le code de retour 0 (succès), ce qui entraîne l'impression de `prêt !` sur la sortie standard.
Une fois la « liste-ET » terminée, le shell peut lancer la commande suivante, affichant `voilà !` sur la sortie standard.

```
$ sleep 5 && echo 'prêt !'; echo 'voilà !'
prêt !
voilà !
```

Autre exemple énervant, si vous avez un éditeur de texte en mode graphique (par exemple `gedit`, `kate`, ...) et que vous voulez le lancer depuis le terminal :

```
$ gedit chef_doeuvre.c
Je ne peux plus
lancer de commandes GRRRRR
```

Vous ne reprendrez la main sur le shell que lorsque vous aurez quitté votre éditeur de texte car le shell *attend* que cette commande se termine !

Non attente de la fin d'une commande

Si on termine la commande `sleep 5 && echo 'prêt !'` par une esperluette (`&`) à la place de `;`, le shell n'attend pas la fin de cette commande pour lancer la suivante.

```
$ sleep 5 && echo 'prêt !'& echo 'voilà !'
[1] 7829
voilà !
$ prêt !
```

Non attente de la fin d'une commande

Si on termine la commande `sleep 5 && echo 'prêt !'` par une esperluette (`&`) à la place de `;`, le shell n'attend pas la fin de cette commande pour lancer la suivante.

```
$ sleep 5 && echo 'prêt !'& echo 'voilà !'
[1] 7829
voilà !
$ prêt !
```

- Le shell a imprimé deux nombres : le premier, entre crochets est le *numéro de job* de la commande en arrière-plan et le second est son *PID* (*Process ID*, c'est-à-dire identifiant de processus).
- L'affichage de `prêt !` qui faisait partie de la commande en arrière-plan peut se faire de façon assez désordonnée... (ici juste après l'invite de commande, mais pas d'inquiétude, cela ne gêne pas pour entrer de nouvelles commandes).

Non attente de la fin d'une commande

```
$ firefox&
[1] 8912
$ gedit idees_geniales.tex& evince idees_geniales.pdf &
[2] 8914
[3] 8915
$ pdflatex idees_geniales.tex
This is pdfTeX, Version 3.14159265-2.6-1.40.20 (TeX Live
2019) (preloaded format=pdflatex)
...
```

Tant que ces commandes ne sont pas terminées :

- `firefox` a pour *numéro de job* 1 et pour *PID* 8841 ;
- `gedit idees_geniales.tex` a pour *numéro de job* 2 et pour *PID* 8914 ;
- `evince idees_geniales.pdf` a pour *numéro de job* 3 et pour *PID* 8915 ;

Non attente de la fin d'une commande

```
$ firefox&
[1] 8912
$ gedit idees_geniales.tex& evince idees_geniales.pdf &
[2] 8914
[3] 8915
$ pdflatex idees_geniales.tex
This is pdfTeX, Version 3.14159265-2.6-1.40.20 (TeX Live
2019) (preloaded format=pdflatex)
...
```

Tant que ces commandes ne sont pas terminées :

- `firefox` a pour *numéro de job* 1 et pour *PID* 8841 ;
- `gedit idees_geniales.tex` a pour *numéro de job* 2 et pour *PID* 8914 ;
- `evince idees_geniales.pdf` a pour *numéro de job* 3 et pour *PID* 8915 ;

Le processus `pdflatex idees_geniales.tex` a un numéro de job (sans doute 4 ?) et un *PID* (péeut-être 8916 ?) mais pour l'instant on ne les connaît pas.

Le paramètre spécial $!

Le paramètre spécial `$!` se développe en le *PID* du dernier processus lancé en arrière-plan.

```
$ sleep 1000&
[1] 13155
$ echo $!
13155
$ sleep 1337&
[2] 13183
$ echo $!
13183
```

Le terminateur de commande &

- Une commande terminée par le caractère « esperluette » (`&`) est exécutée *en arrière-plan*.
- Cela signifie que le shell *n'attend pas* la fin de la commande avant de lancer la suivante (ou d'être prêt à recevoir la commande suivante de l'utilisateur).

On l'utilise souvent pour :

- des commandes qui ouvrent des fenêtres dans l'environnement graphique (`$ firefox &` ou `$ gedit&` par exemple) et qui ne seront donc pas terminées avant que l'utilisateur ne les quitte manuellement.
- des commandes qui prennent beaucoup de temps à se terminer (par exemple la compilation d'un gros programme `$ make &` ou de gros calculs `$ ./gros_calc&`).

Une commande lancée par un shell interactif a un *numéro de job pour ce shell* (et un *PID* comme tous les processus).

Scripts shell : construction `for`

Processus : (non-)définition

Les paramètres positionnels

Manipuler les paramètres positionnels

Exécution d'une commande *en arrière-plan*

Job control

Arrêter ou suspendre une commande au premier plan

Après la première commande **sleep**, on a tapé sur **Ctrl-c** et après la suivante sur **Ctrl-z**.

```
$ sleep 60000 # oups !
^C
$ sleep 60
^Z
[1]+  Stoppé                 sleep 60
```

Arrêter ou suspendre une commande au premier plan

Après la première commande `sleep`, on a tapé sur `Ctrl-c` et après la suivante sur `Ctrl-z`.

```
$ sleep 60000 # oups !
^C
$ sleep 60
^Z
[1]+  Stoppé                  sleep 60
```

Depuis le terminal,

- `Ctrl-c` envoie le signal `SIGINT` (*interrupt*) au processus au premier plan, ce qui en général le *tue* ;
- `Ctrl-z` envoie le signal `SIGTSTP` (*terminal stop*) au processus au premier plan, ce qui en général le met en *pause*.

Pour le processus ayant reçu `SIGTSTP`, le shell imprime son *numéro de job*, suivi du signe `+` et de *l'état* de ce processus : il est *stoppé*.

Jobs

La commande interne `jobs` liste les jobs en cours pour le shell interactif. Les seules commandes que l'on pourra voir sont [1] :

- les commandes en arrière-plan ;
- les commandes stoppées (par exemple avec `Ctrl-z`) ;
- les commandes terminées (les processus morts) alors qu'elles étaient stoppées ou en arrière-plan et que le shell n'a pas encore rapportées comme telles.

```
$ firefox& sleep 3&
[1] 18425
[2] 18426
$ sleep 100
^Z
[3]+  Stoppé                  sleep 100
$ jobs
[1]   En cours d'exécution    firefox &
[2]-  Fini                     sleep 3
[3]+  Stoppé                  sleep 100
```

1. si une commande s'exécute au premier plan, on ne peut pas lancer la commande `jobs`...

`bg` et `fg`

- La commande `bg` (*background*, arrière-plan) reprend l'exécution d'un job stoppé en le mettant en arrière-plan ;
- La commande `fg` (*foreground*, premier plan) reprend l'exécution d'un job stoppé en le mettant au premier plan.

Ces commandes prennent comme argument un numéro de job, précédé de `%`, ou bien sans argument s'appliquent au *job courant*, qui est le dernier job stoppé, s'il y en a, sinon le dernier job lancé en arrière-plan.

bg et fg

```
$ gedit
GRR j'ai oublié de le mettre en arrière-plan !!!!
^Z
[1]+  Stoppé                 gedit
$ bg
$ jobs
[1]+  En cours d'exécution   gedit &
$ sleep 100&
[2] 21089
$ jobs
[1]-  En cours d'exécution   gedit &
[2]+  En cours d'exécution   sleep 100 &
$ fg
sleep 100
^Z
$ bg %2
[2]+ sleep 100 &
```

`bg` et `fg`

Utilisation courante de `fg` :

- Exécution d'une commande qui prend toute la fenêtre du terminal, par exemple un éditeur visuel dans le terminal comme `vi`, `nano` ou `emacs -nw` ou bien les commandes `man`, `less`, ...
- Mise en pause de la commande avec `Ctrl-z` ; retour au terminal pour, par exemple, compiler un programme, essayer une commande, etc.
- Commande `fg` (sans argument) pour revenir au programme précédent (par exemple l'éditeur de texte), dans l'état où on l'avait laissé.