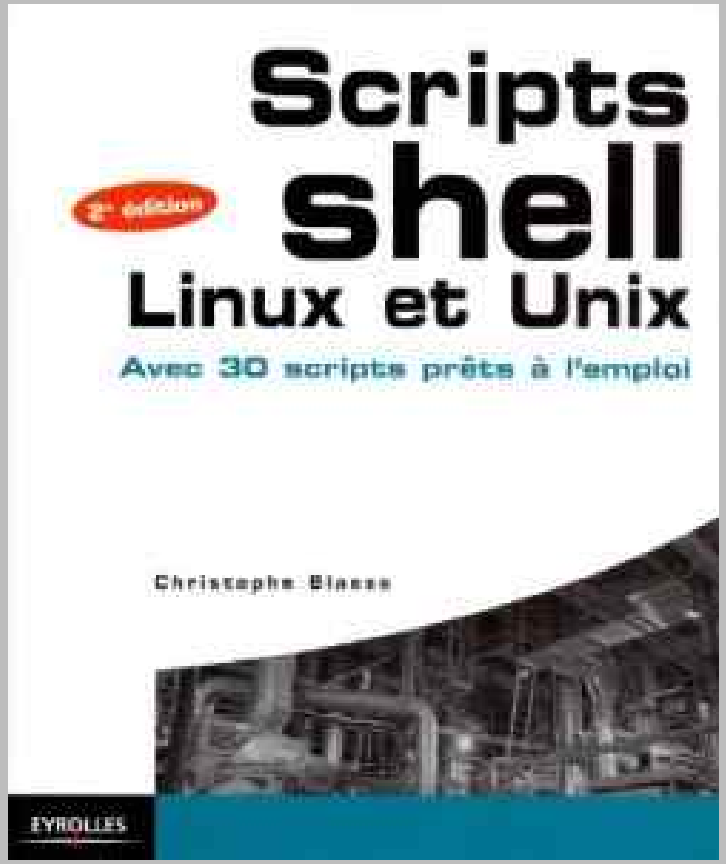
Scripts shells - Linux

Saïd El Mamouni – Janvier 2019

Une référence



Un shell Unix est un **interpréteur de commandes** destiné aux systèmes d'exploitation Unix et de type Unix.

- Il permet d'accéder aux fonctionnalités internes du système d'exploitation.
- Il se présente sous la forme d'une interface en ligne de commande accessible depuis la console ou un terminal.
- L'utilisateur lance des commandes sous forme d'une entrée texte exécutée ensuite par le shell. Dans les différents systèmes d'exploitation Microsoft Windows, le programme analogue est command.com, ou cmd.exe.
- Les systèmes d'exploitation de type Unix disposent le plus souvent d'un shell.

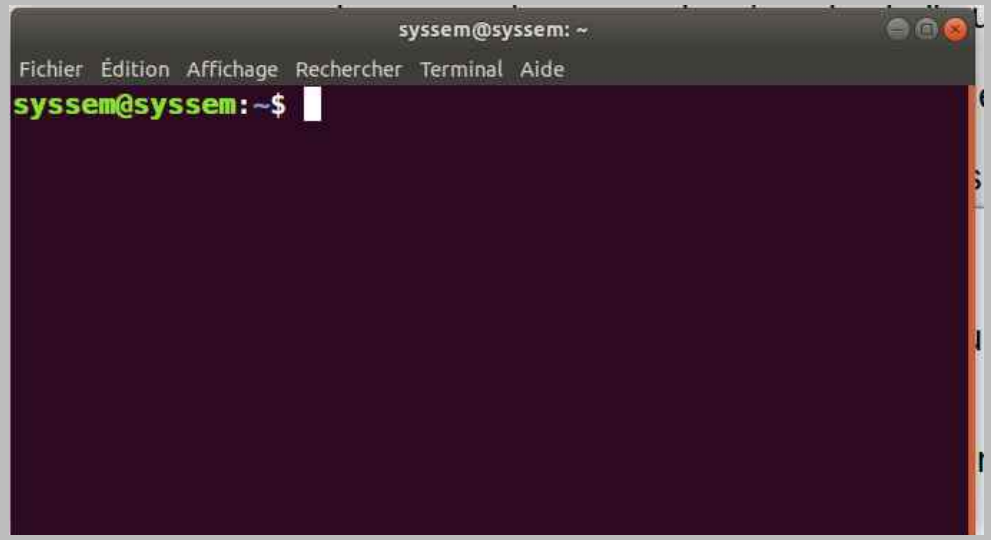
Alacritty

rxvt-unicode

Terminology

Gnome Terminal

Eterm



Xfce4 terminal

Konsole

Wterm

Yakuake

Termite

Tilix

Guake

Sakura

ROXTerm

st

LilyTerm

Tilda

tmux

QTerminal

I - Historique

Le premier shell est le Thompson shell (en) apparu en 1971 avec la première version d'Unix et écrit par Ken Thompson, l'un des créateurs d'Unix. Il est remplacé par le Bourne shell, écrit par Stephen Bourne, en 1977 pour la version 7 d'Unix.

En 1978, Bill Joy, alors étudiant à l'Université de Californie à Berkeley, crée csh (C shell), une évolution du shell dont la syntaxe s'inspire de celle du langage C. Il permet notamment la réutilisation de l'historique des commandes. Une version plus moderne du csh est ensuite publiée sous le nom tcsh.

Le Korn shell (ksh) est publié en 1983 par David Korn. Il est compatible avec le Bourne shell, reprend certaines fonctionnalités de csh et ajoute des fonctions de scripts avancées disponibles dans des langages plus évolués tels que le Perl.

Le Bourne-Again shell (bash) apparait quant à lui en 1988. Il est écrit par Brian Fox pour la Free Software Foundation dans le cadre du projet GNU. C'est le shell de nombreuses implémentations libres d'Unix, telles que les systèmes GNU/Linux. Il est compatible avec le Bourne shell dont il se veut une implémentation libre.

Paul Falstad crée Zsh en 1990 alors qu'il est étudiant à l'université de Princeton. Ce shell reprend les fonctions les plus pratiques de bash, Csh, tcsh.

En 2005 apparaît le fish (Friendly Interactive Shell).
C’est un shell Unix adapté pour une utilisation interactive. Ses caractéristiques sont ciblées sur la convivialité et la découverte.

La syntaxe de fich est simple mais incompatible avec d’autres langages shell. L'utilisation de couleur à même l'invite et dans les autocomplétions rend la saisie plus facile. Plusieurs différences notables rendent fish bien plus ergonomique que le shell bash.

II - Installer un nouveau shell

L'installation d'un nouveau shell se fait de façon standard, comme n'importe quel paquet :

```
# apt-get install ksh zsh tcsh
```

Une fois installé, il faut demander à l'utiliser pour votre compte utilisateur.

```
$ chsh
Mot de passe :
```

Changement d'interpréteur de commandes initial pour syssem
Entrez la nouvelle valeur ou « Entrée » pour conserver la valeur proposée
Interpréteur de commandes initial [/bin/bash]:

Pour être interprétée, un fichier bash doit commencer par la ligne suivante :

#!/bin/bash

- Le **#!** est appelé le sha-bang, (on écrit aussi shebang).
- **/bin/bash** peut être remplacé par /bin/sh si vous souhaitez coder pour sh, /bin/ksh pour ksh, etc.

Bien que non indispensable, cette ligne permet de s'assurer que le script est bien exécuté avec le bon shell.

Si la ligne manque, c'est le shell de l'utilisateur qui sera chargé et cela peut poser des dysfonctionnement.

La ligne du sha-bang permet donc de « charger » le bon shell avant l'exécution du script.

Pour pouvoir être lancé dans un terminal le fichier doit être rendu exécutable.

```
$chmod 700 mon_fichier_bash.sh
$chmod +x   mon_fichier_bash.sh
```

Le script peut alors être lancé de cette facon :

```
$./mon_fichier_bash.sh
```

Ou en donnant le chemin entier

```
$/home/syssem/scripts/mon_fichier_bash.sh
```

Ou en appelant l'interprète bash simplement

```
$bash mon_fichier_bash.sh
```

```
$bash -x mon_fichier_bash.sh  (en mode débogage)
```

Ou en ajoutant le répertoire ou se trouve le fichier dans le **PATH**.

```
PATH=$PATH:/home/syssem/
```

Le PATH est une variable système qui indique où sont les programmes exécutables.

```
$echo $PATH
```

renvoi la liste de ces répertoires « spéciaux ».

Attention : La syntaxe bash est très rigoureuse. Notamment, les espaces sont interprétés.

III - Afficher et manipuler des variables

Comme dans tous les langages de programmation, bash utilise des variables. Elles permettent de stocker temporairement des informations en mémoire.

C'est la base de la programmation !

Exemple avec le fichier variable.sh :

```
#!/bin/bash

message='Hello world'
```

Variable : message
Valeur de la variable : Hello world

Pas d'espaces autour du symbole égal « = »

Si nous avions voulu insérer une apostrophe dans la valeur de la variable, nous l'aurions faîtes précédé d'un antislash \.

Comme les apostrophes servent à délimiter le contenu, il faut utiliser un caractère d'échappement

```
#!/bin/bash

message='Coucou c\'est moi'
```

Pour faire afficher le message à l'écran il faut utiliser la commande **echo**

```
#!/bin/bash

echo message='Coucou c\'est moi'
```

IV – Variables d'environnement¶

On en distingue trois types : utilisateur, système et spéciales. Le principe est de pouvoir affecter un contenu à un nom de variable, généralement une chaîne de caractère ou des valeurs numériques.

Certaines variables ont une signification spéciale réservée. Ces variables sont très utilisées lors la création de scripts :

* pour récupérer les paramètres transmis sur la ligne de commande,
* pour savoir si une commande a échoué ou réussi,
* pour automatiser le traitement de tous paramètres.

Afficher les variables affectés à son environnement

```
$env ou $printenv
```

Ajouter le chemin vers le dossier des scripts dans son environnement.

$PATH=$PATH:/home/syssem/SCRIPTS/

Lancer le script avec la commande bash

$bash script.sh

```
syssem@syssem:~$ env

SHELL=/bin/bash
XDG_MENU_PREFIX=xfce-
MANDATORY_PATH=/usr/share/gconf/xubuntu.mandatory.path
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
DESKTOP_SESSION=xubuntu
SSH_AGENT_PID=2095
PWD=/home/syssem
LOGNAME=syssem
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
XAUTHORITY=/run/user/1000/gdm/Xauthority
HOME=/home/syssem
USERNAME=syssem
LANG=fr_FR.UTF-8
XDG_CURRENT_DESKTOP=XFCE
LESSCLOSE=/usr/bin/lesspipe %s %s
XDG_SESSION_CLASS=user
TERM=xterm-256color
GTK_OVERLAY_SCROLLING=0
DEFAULTS_PATH=/usr/share/gconf/xubuntu.default.path
LESSOPEN=| /usr/bin/lesspipe %s
LIBVIRT_DEFAULT_URI=qemu:///system
USER=syssem
DISPLAY=:0.
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

Quelques variables standard maintenues automatiquement:

LOGNAME: nom de l'utilisateur actuel
HOME: répertoire du compte de l'utilisateur actuel
TERM: type de terminal utilisé
PS1: comment afficher l'indicatif (le prompt)
SHELL: où se trouve le shell en cours
PATH: dans quels répertoires chercher les commandes externes
PWD: répertoire actuel
0: chemin complet de votre programme
1, 2, 3...: paramètres donnés à votre programme
#: nombre de paramètres donnés à votre programme
$: numéro du processus du shell
?: dernière valeur retournée par un programme.

Par convention, les variables destinées à l'environnement sont en **majuscules et les autres en minuscules.**

Pour obtenir la longueur d'une chaîne stockée dans une variable, on écrit **${#VAR}.**

Liste de variables prépositionnées¶

$0 : nom du script.
Plus précisément, il s'agit du paramètre 0 de la ligne de commande, équivalent de argv[0]

$1, $2, ..., $9 : respectivement premier, deuxième, ..., neuvième paramètre de la ligne de commande

$* : tous les paramètres vus comme un seul mot

$@ : tous les paramètres vus comme des mots séparés : "@" équivaut à "1" "$2" …

$# : nombre de paramètres sur la ligne de commande

$- : options du shell

$? : code de retour de la dernière commande

Valeurs de vérité

\$\$: PID du shell

$! : PID du dernier processus lancé en arrière-plan

$_ : dernier argument de la commande précédente

Valeurs de vérité

Chaque commande retourne une valeur qui détermine si le programme a bien fonctionné ou non.
C'est la raison du **return 0**; dans un programme C: **0** indique que tout s'est bien passé.

Une commande qui retourne autre chose que **0** a rencontré un problème. La valeur de retour de la dernière commande exécutée se trouve dans la variable **?**.

Exemple :

rm fichier
Echo $?

Un script peut être appelé avec des paramètres :

```
$ mon-script param1 param2 param3
```

Les paramètres sont alors placés dans un vecteur de variables :
- param1 dans la première case du vecteur,
- param2 dans la deuxième case,
- param3 dans la troisième case, etc.

Vous pouvez accéder aux éléments du vecteur grâce à des variables $1, $2 et ainsi de suite.
La variable donnant la longueur du vecteur est $#.

```
#!/bin/bash
# script1.sh
echo "Nom du script $0"
echo "premier paramètre $1"
echo "second paramètre $2"
echo "PID du shell " ${$}
echo "code de retour $?"
exit
```

```
$chmod +ux script1.sh
```

```
$./script1.sh 10 zozo
```

```
$ echo ${$}                          # Quel est le pid du shell courant ?
103
```

```
$ ps | grep 103                      # Cela correspond à bash ?
  103 pts/1    00:00:00 bash
```

PID

echo ${$}

$vim testpid.sh

```
#!/bin/bash
echo ${$}
sleep 10 ;
```

```
$ ./testpid.sh &                                  # Exécution du script :
[2] 1467
1467
```

```
$ ps | grep 1467                     # Est-ce bien le pid du script ?
 1467 pts/1    00:00:00 testpid.sh
```

1.5. Variables internes¶

En début de script, on peut définir la valeur de départ des variables utilisées dans le script.

```
VARIABLE="valeur"
```

Elles s'appellent comme ceci dans le script :

```
echo $VARIABLE
```

Il peut être utile de marquer les limites d'une variable avec les accolades.

```
echo ${VARIABLE}
```

Après le sha-bang, nous pouvons commencer à coder.

Le principe est très simple : il vous suffit d'écrire les commandes que vous souhaitez exécuter. Ce sont les mêmes que celles que vous tapiez dans l'invite de commandes !

ls : pour lister les fichiers du répertoire.

cd : pour changer de répertoire.

mkdir : pour créer un répertoire.

grep : pour rechercher un mot.

sort : pour trier des mots.

etc.

Listes (enchaînement de commandes)

On peut composer les commandes en listes. Les opérateurs de composition sont:

commande1; commande2:
→ exécuter la commande 1 puis la commande 2 Ceci est équivalent à entrer deux commandes sur deux lignes, mais est plus compact

commande1 & commande2:
→ exécuter commande 1, la mettre en arrière-plan et exécuter la commande 2 en même temps

Commande 1 && commande 2:
→ n'exécuter la commande 2 seulement si la commande 1 a réussi (ie, a retourné 0)

commande1 || commande2:
→ n'exécuter la commande 2 seulement si la commande 2 a échoué (ie, a retourné autre chose que 0)

Premiers scripts simple :

```
#!/bin/bash
# script2.sh
PRENOM="francois"
echo "dossier personnel /home/$PRENOM"
exit
```

```
$ chmod +x script2.sh
$ ./script2.sh
$ dossier personnel /home/francois
```

La commande echo permet simplement d'afficher une ligne

```
#!/bin/bash
test(){
   echo "Bonjour"
}
test
```

```
$ chmod +x test
$ ./test
$ Bonjour
```

Quelques utilisations pratiques

Pour écrire à la fin d'un fichier sans en écraser le contenu, on utilise les signes >> :

```
$ echo "Ma jolie phrase est belle." >> monfichier
```

Pour écraser un fichier, en effaçant tout son contenu, on utilise le signe > :

```
$ echo "Ma jolie phrase est belle." > monfichier
```

Si vous utilisez “sudo” pour obtenir les droits root (comme sur ubuntu par exemple), et que vous vouliez utiliser “echo” pour écrire dans un fichier qui appartient à root, vous devrez ruser car le “sudo” ne survivra pas à la redirection. Ça sera plus clair avec un exemple :

Ne fonctionnera PAS :

```
$ sudo echo 'www.nouveau_dépôt' >> /etc/apt/sources.list
```

Vous obtiendrez un refus avec “bash: /etc/apt/sources.list: Permission non accordée” comme motif. Pour que ça fonctionne il faut utiliser un autre type de redirection, par exemple avec la commande “tee” qui sert justement à ça.

Vous obtiendrez un refus avec “bash: /etc/apt/sources.list: Permission non accordée” comme motif. Pour que ça fonctionne il faut utiliser un autre type de redirection, par exemple avec la commande “tee” qui sert justement à ça.

Vous pouvez utiliser :

```
$ echo 'www.nouveau_dépôt' | sudo tee -a /etc/apt/sources.list
```

:!: L'option “-a” indique à “tee” d'ajouter la ligne en fin de fichier, sinon <color red>le comportement par défaut de “tee” est de remplacer le fichier cible</color>. Ne pas l'oublier !

Pour écrire plus d'une ligne avec “echo”, vous pouvez utiliser un saut de ligne, noté “\n”. Pour indiquer à “echo” que ce symbole doit être interprété comme un saut de ligne, il faut utiliser l'option “-e” :

```
$ echo -e '#ceci est un commentaire \nma deuxième ligne' >> /home/tux/test.txt
```

Vous obtiendrez dans le fichier “/home/tux/test.txt” le résultat suivant:

```
#ceci est un commentaire
ma deuxième ligne
```

Pratique pour ajouter une ligne d'option dans un fichier de configuration, et un commentaire explicatif en même temps.

Test simple

L'instruction **if** permet d'effectuer des opérations si une condition est réalisée.
l'expression conditionnelle est exécutée si son code de retour vaut zéro.

```
if      condition
then
        instruction(s)
fi
```

Test avancé

L'instruction **if** peut inclure une instruction **else** permettant d'exécuter des instructions dans le cas ou la condition n'est pas réalisée.

```
if      condition
then
        instruction(s)
else
        instruction(s)
fi
```

Avec des **if** imbriqué dans d'autres **if**

```
if      condition1
        then
                instruction(s)
        else
                if      condition2
                then
                        instruction(s)
                        else
                        if      condition3
                                ...
                        fi
                fi
        fi
```

ksh fournit un raccourci d'écriture : **elif.**
Le code précédent pourrait être réécrit ainsi :

```
if      condition1
then
        instruction(s)
elif    condition2
then
        instruction(s)
elif    condition3
                ...
fi
```

L'instruction **case** permet de comparer une valeur avec une liste d'autres valeurs et d'exécuter un bloc d'instructions lorsque une des valeurs de la liste correspond.

```
case valeur_testee in
        valeur1) instruction(s);;
        valeur2) instruction(s);;
        valeur3) instruction(s);;
        ...
        esac
```

Équivalent avec if

```
if [ valeur_teste = valeur1 ]
   then instruction(s)
elif [ valeur_testee = valeur2 ]
   then instruction(s)
elif [ valeur_testee = valeur3 ]
   then instruction(s)
...
fi
```

La commande test

Cette commande permet de faire de comparaison utiles:

test string1 = string2 # retourne vrai si les chaînes a et b sont identiques
test string1 != string2 # vrai si différentes
test -e fichier # vrai si fichier existe (fichier normal, un répertoire, un lien ou autre chose)
test -f fichier # vrai si le fichier existe et est un fichier normal
test -r fichier # vrai si le fichier est lisible
test -x fichier # vrai si le fichier est un programme exécutable

test $nb1 OP $nb2 # vrai si le nombre 1 OP le nombre 2, où OP est:
-eq (equal), -ne (not equal), -lt (less than),
-le (less than or equal), -gt (greater than), ou
-ge (greater than or equal)

test ! expr # vrai si expr est fausse
test -n string # vrai si string est de longueur supérieure à 0
test -z string # vrai si string est de longueur 0 (ex.: test -z "")

En général on n'écrit cependant jamais "test expr". On utilise plutôt la forme condensée [expr].

Évaluation arithmétique

Pour évaluer arithmétiquement une expression, la mettre entre parenthèses doubles.

(($a > $b)) # Retourne vrai si a plus grand que b, faux sinon

sum=$(($a + $b)) # En autant que a et b contiennent des nombres

Les mêmes opérateurs sont disponibles qu'en C, avec les mêmes règles de précédence. Il y a aussi l'opérateur d'exponentiation, représenté par **.

Fonctions
On peut définir des fonctions en bash; cela est très utile pour structurer ses programmes et fait tr&eagrave;s plaisir à votre démonstrateur:

nom_de_fonction() {
corps
}

Pour appeler la fonction, simplement:

nom_de_fonction param1 param2 param3

À l'intérieur du corps de la fonction, les paramètres sont disponibles dans les variables **$1, $2, $3, etc**.
Le nombre de paramètres est dans la variable **$#**.

Ne pas oublier qu'une variable utilisée à l'intérieur d'une fonction est globale! Pour éviter ce phénomène, il faut dire au shell si une variable doit être locale. Pour ce faire, utiliser la commande interne:

local nomvar

Pour sortir d'une fonction, utiliser la commande interne **return**; la fonction va sortir avec la valeur de retour de la dernière commande exécutée. Si ce n'est pas ce qui est désiré, utiliser **return N**.

Un script peut être appelé avec des paramètres :

```
$ mon-script param1 param2 param3
```

Les paramètres sont alors placés dans un vecteur de variables :
- param1 dans la première case du vecteur,
- param2 dans la deuxième case,
- param3 dans la troisième case, etc.

Dans le fichier du script vous pouvez accéder aux éléments du vecteur grâce à des variables :
- la variable correspondant au premier élément du vecteur est $1
- celle correspondant au deuxième élément est $2 et ainsi de suite.

La variable donnant la longueur du vecteur est $#.

Interaction utilisateur¶

La commande **echo** pose une question à l'utilisateur

La commande **read** lit les valeurs entrées au clavier et les stocke dans une variable à réutiliser.

```
echo "question"
read reponse
echo $response
```

On peut aller plus vite avec **read -p** :

```
# !/bin/bash
read -p "question" reponse
echo $reponse
```

```
$ ./qr.sh
$ posez votre question : quelle heure est il ?
10h
```

????

Excercice :

Ecrire un script bash qui indique si un argument (fichier ou repertoire) donné en paramètre existe ou n'existe pas.

```
#!/bin/bash

name=$1
if [ "$name" = "" ]
then
        echo "Entrez un argument"
else
        if [ -e "$name" ]
        then
                echo "$name existe"
        else
                echo "$name n'existe pas"
        fi
fi
```

https://fr.wikibooks.org/wiki/Programmation_Bash/Scripts#Au_commencement,_la_ligne_shebang

Excercice :

1. on veut faire un copie de tous les fichiers du repertoire courant sous le nom *.bak

```
#!/bin/bash

for i in ./* ; do
        cp "$i" "$i.bak"
Done
```

https://fr.wikibooks.org/wiki/Programmation_Bash/Scripts#Au_commencement,_la_ligne_shebang

Excercice :

2. chercher si les arguments existent

```
#!/bin/bash

if [ "$1" == "" ]
then
        echo "please enter one argument"
else

        for i in "$@"
do

        if [ -e $i ]
        then
                echo "$i exists"
        else
                echo "$i does not exist"
        fi

done
fi
```

https://fr.wikibooks.org/wiki/Programmation_Bash/Scripts#Au_commencement,_la_ligne_shebang

https://fr.wikipedia.org/wiki/Langage_de_script

https://docplayer.fr/16163428-Les-modules-sisr-de-deuxieme-annee.html

https://linux.goffinet.org/08-scripts-shell/

https://openclassrooms.com/fr/courses/43538-reprenez-le-controle-a-laide-de-linux/43126-afficher-et-manipuler-des-variables

https://openclassrooms.com/forum/sujet/exercices-bash-57809#r4438739