

FORMATUX

Shell Bash

Antoine Le Morvan, Xavier Sauvignon

Version 2.0.2 du 24 septembre 2019

Table des mati•res

PrŽface	2
CrŽdits	2
L'histoire de Formatux	3
Licence	4
Comment contribuer au projet ?	4
Antora	6
Gestion des versions	6
1. Les scripts shell - Niveau 1	7
1.1. Premier script	8
1.2. Variables	10
1.3. Saisie et manipulations	16
Tester vos connaissances	24
2. Les scripts shell - Instructions de contr™le	26
2.1. Les tests	26
2.2. Structures conditionnelles	33
2.3. Boucles	38
Tester vos connaissances	43
3. TP Scripting shell	46
3.1. ftude du besoin	46
3.2. Consignes	47
3.3. Pistes de travail	47
3.4. Proposition de correction	47
Glossaire	56
Index	57

Le scripting BASH sous LINUX.

Dans cette partie, vous allez pouvoir vous initier au scripting Bash, exercice que tout administrateur devra réaliser un jour ou l'autre.

Un TP Final vous est proposé avec des pistes de réalisation.

Préface

GNU/Linux est un système d'exploitation libre fonctionnant sur la base d'un noyau Linux, également appelé kernel Linux.

Linux est une implémentation libre du système UNIX et respecte les spécifications POSIX.

GNU/Linux est généralement distribué dans un ensemble cohérent de logiciels, assemblés autour du noyau Linux et prêt à être installé. Cet ensemble porte le nom de «Distribution».

¥ La plus ancienne des distributions est la distribution Slackware.

¥ Les plus connues et utilisées sont les distributions Debian, RedHat et Arch, et servent de base pour d'autres distributions comme Ubuntu, CentOS, Fedora, Mageia ou Manjaro.

Chaque distribution présente des particularités et peut être développée pour répondre à des besoins très précis :

¥ services d'infrastructure ;

¥ pare-feu ;

¥ serveur multimédia ;

¥ serveur de stockage ;

¥ etc.

La distribution présentée dans ces pages est principalement la CentOS, qui est le pendant gratuit de la distribution RedHat, mais la plupart des supports s'appliquent généralement aussi à Ubuntu et Debian. La distribution CentOS est particulièrement adaptée pour un usage sur des serveurs d'entreprises.

Crédits

Ce support de cours a été rédigé sur plusieurs années par de nombreux formateurs.

Les formateurs à l'origine du projet et contributeurs principaux actuels :

¥ Antoine Le Morvan ;

¥ Xavier Sauvignon ;

Notre relecteur principal des premières versions :

¥ Patrick Finet ;

Il a rédigé la partie Git :

¥ Carl Chenet

Ils ont contribués à la rédaction :

¥ Nicolas Kovacs : Apache sous CentOS7 ;

¥ Damien Dubédat : Scripting shell ;

¥ É

Enfin, les illustrations de qualité sont dessinées pour formatux par François Muller (aka [Founet](#)).

L'histoire de Formatux

Nous étions (Xavier, Antoine et Patrick) tous les trois formateurs dans la même école de formation pour adulte.

L'idée de fournir aux stagiaires un support en PDF reprenant la totalité des cours dispensés pour leur permettre de réviser et approfondir les points vus en classe pendant la journée nous a rapidement paru être une priorité.

En décembre 2015, nous testions les solutions qui existaient pour rédiger un support d'une telle taille, et nous avons retenu dès le début du projet le format AsciiDoc pour sa simplicité et le générateur AsciiDoctor pour la qualité du support généré, la possibilité de personnaliser le rendu mais surtout pour l'étendue de ses fonctionnalités. Nous avons également testé le markdown, mais avons été plus rapidement limités.

!

5 ans après le début du projet et après avoir basculé notre site web sous Antora, nous ne regrettons absolument pas le choix technique d'AsciiDoc.

La gestion des sources a été confiée dès l'année suivante à la forge Gitlab de Framagit, ce qui nous permettait de travailler à plusieurs en même temps sur le support, et de faciliter la relecture du support par Patrick. En découvrant la CI de gitlab-ci, nous avons enfin la stack qui nous permettrait d'automatiser totalement la génération du support. Il ne nous manquait plus que la génération d'un site web depuis ces mêmes sources.

Le travail de rédaction étant fortement personnel, sachant que nous risquions d'être mutés rapidement dans les années à suivre, nous avons voulu ce support sous Licence Libre, afin qu'un maximum de personnes puissent à la fois contribuer et en profiter, et que notre beau support ne se perde pas.

Même après avoir tous quitté notre organisme de formation, nous avons continué à faire vivre le projet formatux, en faisant évoluer certaines parties, en nous appuyant sur d'autres pour nos formations et en intégrant de nouvelles parties, comme la partie Git de Carl Chenet.

En juillet 2019, nous (Xavier et Antoine, Patrick ayant pris sa retraite informatique) avons décidé de reprendre le développement de Formatux plus activement et d'en faire un peu plus sa promotion. L'organisation complète du support a été revue, en le scindant en 8 dépôts distincts, correspondant à chacune des parties, au support global ainsi qu'au site web. Nous avons voulu notre organisation full devops, afin que la génération de chacune des parties soient totalement automatisées et inter-

dépendantes les unes des autres.

Il est difficile aujourd'hui d'évaluer la popularité de notre support. Ce dernier a longtemps été disponible en téléchargement par torrent sur freetorrent (malheureusement aujourd'hui disparu) et en direct depuis le site web. Nous n'avons pas de métriques et nous n'en voulons pas particulièrement. Nous retirons notre satisfaction dans les contacts que nous avons avec nos lecteurs.

Licence

Formatux propose des supports de cours Linux à destination des formateurs ou des personnes désireuses d'apprendre à administrer un système Linux en autodidacte.

Les supports de Formatux sont publiés sous licence Creative Commons-BY-SA et sous licence Art Libre. Vous êtes ainsi libre de copier, de diffuser et de transformer librement les œuvres dans le respect des droits de l'auteur.

BY : Paternité. Vous devez citer le nom de l'auteur original.

SA : Partage des Conditions Initiales à l'identique.

✎ Licence Creative Commons-BY-SA : <https://creativecommons.org/licenses/by-sa/3.0/fr/>

✎ Licence Art Libre : <http://artlibre.org/>

Les documents de Formatux et leurs sources sont librement téléchargeables sur [formatux.fr](https://www.formatux.fr) :

✎ <https://www.formatux.fr/>

Les sources de nos supports sont hébergées chez Framasoft sur leur forge Framagit. Vous y trouverez les dépôts des codes sources à l'origine de la version de ce document :

✎ <https://framagit.org/formatux/>

A partir des sources, vous pouvez générer votre support de formation personnalisé. Nous vous recommandons le logiciel AsciiDocFX téléchargeable ici : <http://asciidocfx.com/> ou l'éditeur Atom avec les plugins AsciiDoc.

Comment contribuer au projet ?

Si vous voulez participer à la rédaction des supports formatux, forkez-nous sur framagit.org.

Vous pourrez ensuite apporter vos modifications, compiler votre support personnalisé et nous proposer vos modifications.

Vous êtes les bienvenus pour :

✎ Compléter le document avec un nouveau chapitre,

-
- ¥ Corriger ou compléter les chapitres existants,
 - ¥ Relire le document et corriger l'orthographe, la mise en forme,
 - ¥ Promouvoir le projet

De votre côté

1. Créer un compte sur <https://framagit.org>,
2. Créer un fork du projet que vous voulez modifier parmi la liste des projets du groupe : [Créer le fork](#),
3. Créer une branche nommée develop/[Description],
 - ! O• [Description] est une description très courte de ce qui va être fait.
4. Faire des commits dans votre branche,
5. Pusher la branche sur votre fork,
6. Demander une merge request.

!

Si vous n'êtes pas un grand utilisateur de git, ce n'est pas grave. Vous pouvez toujours lire la partie Git de notre support puis nous contacter. Nous vous guiderons ensuite pour vos premiers pas.

Cette remarque est également vraie pour le format AsciiDoc.

Essayer de conserver le même ton qui est déjà employé dans le reste du support (pas de 'je' ni de smileys par exemple).

La suite se passe de notre côté

1. Quelqu'un relira votre travail,
 - ! Essayez de rendre ce travail plus facile en organisant vos commits.
2. S'il y a des remarques sur le travail, le relecteur fera des commentaires sur la merge request,
3. Si la merge request lui semble correcte il peut merger votre travail avec la branche develop.

Corrections suite à une relecture

La relecture de la merge request peut vous amener à faire des corrections. Vous pouvez faire ces corrections dans votre branche, ce qui aura pour effet de les ajouter à la merge request.

Comment compiler mon support formatux ?

Après avoir forké notre projet ou l'avoir cloné (`git clone https://framagit.org/group/formatux-PARTIEXX.git`), déplacez-vous dans le dossier formatux-PARTIEXX nouvellement créé.

Vous avez ensuite plusieurs possibilités :

¶ Vous utilisez le logiciel AsciiDocFX ou Atom avec ses plugins AsciiDoc (recommandé sous Windows) : lancez le logiciel, ouvrez le fichier `.adoc` désiré, et cliquez sur le bouton PDF (AsciiDoctorFX) ou regardez la prévisualisation (Atom).

¶ Vous êtes sous Linux et vous avez déjà installé le paquet `asciidoctor` : exécutez la commande `asciidoctor-pdf -t -D public -o support.pdf support.adoc`.

Comment faire un don

Vous pouvez faire un don par paypal pour soutenir nos efforts. Ces dons serviront à payer notre nom de domaine et notre hébergement. Nous pourrions également reverser une partie de ces dons à Framasoft, qui héberge gracieusement nos repos et met à disposition un runner qui compile nos supports et notre site web. Enfin, une petite bière entre nous peut également être au programme.

Accédez à la cagnotte paypal : <https://www.paypal.com/pools/c/8hlM1Affp1>.

Nous contacter

Nous sommes disponibles sur github pour échanger autour de formatux, de Linux et de la formation : <https://github.com/formatux-fr/formatux>.

Antora

Pour la génération de notre site web, nous utilisons Antora. Antora nous permet, depuis les mêmes sources, de pouvoir générer le support en PDF et le site web statique en HTML. Le développement d'Antora est sponsorisé par OpenDevise Inc.

Gestion des versions

Table 1. Historique des versions du document

Version	Date	Observations
1.0	Avril 2017	Version initiale.
2.0	Septembre 2019	Passage à <code>antora</code>
2.0.1	Septembre 2019	Relecture du module Shell Niveau 1 Contribution de Damien Dubédat.

Chapitre 1. Les scripts shell - Niveau 1

" Objectifs pđdagogiques

Dans ce chapitre, vous allez apprendre ˆ rđdiger votre premier script, ˆ utiliser des variables et řgalement ˆ interagir avec les utilisateurs.

- # Rđdiger son premier script en bash ;
- # Utiliser des variables ;
- # Interagir avec un utilisateur ;
- # Transformer le contenu dđun fichier ou dđune variable ;
- # Gřrer les arguments dđun script.

\$ script, shell, bash

Connaissances : %

Niveau technique : !

Temps de lecture : 20 minutes

Le shell est lđinterprřteur de commandes de Linux. Cđest un binaire qui ne fait pas partie du noyau, mais forme une couche supplřmentaire, dđ• son nom de "*coquille*".

Il analyse les commandes saisies par lđutilisateur puis les fait exřcuter par le syst•me.

Il existe plusieurs shells, tous partageant des points communs. Lđutilisateur est libre dđutiliser celui qui lui convient le mieux parmi (entre autres) :

- le Bourne-Again shell (**bash**),
- le Korn shell (**ksh**),
- le C shell (**csh**),
- etc.

Le **bash** est prřsent par dřfaut sur les principales distributions Linux. Il se caractřrise par ses fonctionnalitřs pratiques et conviviales.

Le shell est aussi un langage de programmation basique qui, grˆce ˆ quelques commandes dřdiřes, permet :

- lđutilisation de variables,
- lđexřcution conditionnelle de commandes,
- la rřpřtition de commandes.

Les scripts en shell ont lđavantage dđˆtre rřalisables rapidement et de mani•re fiable, sans compilation ni installation de commandes supplřmentaires. Un script shell nđest quđun fichier

texte sans enluminures (gras, italique, etc.).

!

Si le shell est un langage de programmation Ç basique È, il n'en reste pas moins très puissant et parfois plus rapide qu'un mauvais code compilé. Si vous n'êtes pas convaincu, vous pouvez lire, même si l'article commence à dater, l'article suivant : [Entretien avec un débutant en bash](#) de Étienne Dublé. Cela vous permettra de réfléchir à votre façon de coder.

Pour écrire un script shell, il suffit de réunir dans un même fichier texte toutes les commandes nécessaires. En rendant ce fichier exécutable, le shell le lira séquentiellement et exécutera une à une les commandes le comportant. Il est aussi possible de l'exécuter en passant le nom du script comme un argument au binaire `bash`.

Lorsque le shell rencontre une erreur, il affiche un message permettant d'identifier le problème mais continue l'exécution du script. Mais il existe des mécaniques pour stopper l'exécution d'un script en cas de survenance d'une erreur. Les erreurs propres aux commandes sont également affichées à l'écran ou à l'intérieur de fichiers.

Qu'est ce qu'un bon script ? C'est un script :

- ¥ fiable : son fonctionnement est irréprochable même en cas de mauvaise utilisation ;
- ¥ commenté : son code est annoté pour en faciliter la relecture et les futures évolutions ;
- ¥ lisible : le code est indenté à bon escient, les commandes sont agréables, É
- ¥ portable : le code fonctionne sur tout système Linux, gestion des dépendances, gestion des droits, etc.

1.1. Premier script

Pour commencer l'écriture d'un script shell, il est pratique d'utiliser un éditeur de texte gérant la coloration syntaxique.

`vim`, par exemple, est un outil adapté à cela.

Le nom du script devra respecter quelques règles :

- ¥ pas de nom de commandes existantes ;
- ¥ ne contenant que des caractères alphanumériques, c'est-à-dire pas de caractère accentué ni d'espace ;
- ¥ extension en `.sh` pour indiquer qu'il s'agit d'un script shell.

hello-world.sh

```
1 #!/usr/bin/env bash
2 #
3 # Auteur : Antoine Le Morvan
4 # Date : Janvier 2019
5 # Version 1.0.0 : Affiche le texte "Hello world !"
6 #
7
8 # Affiche un texte ^ l'Žcran :
9 echo "Hello world !"
```

Pour pouvoir exŽcuter ce script, en argument du **bash** :

```
$ bash hello-world.sh
Hello world !
```

Ou, plus simplement, apr•s lui avoir donnŽ le droit d'exŽcution :

```
$ chmod u+x ./hello-world.sh
$ ./hello-world.sh
Hello world !
```



Notez que pour exŽcuter le script, celui-ci a ŽtŽ appelŽ avec `./` Ě avant son nom. L'interprŽteur pourra refuser d'exŽcuter un script prŽsent dans le rŽpertoire courant sans indiquer un chemin (ici avec le `./` Ě devant). La commande **chmod** n'est Ě passer qu'une seule fois sur un script nouvellement crŽŽ.

La premi•re ligne Ě Žcrire dans tout script permet d'indiquer le nom du binaire du shell Ě utiliser pour l'exŽcuter. Si vous dŽsirez utiliser le shell **ksh** ou le langage interprŽtŽ **python**, vous remplacerez la ligne :

```
#!/usr/bin/env bash
```

par :

```
#!/usr/bin/env ksh
```

ou par :

```
#!/usr/bin/env python
```

Tout au long de l'écriture, il faudra penser à la relecture du script en utilisant notamment des commentaires :

• une présentation générale, en début, pour indiquer le but du script, son auteur, sa version, son utilisation, etc.

• au cours du texte pour aider à la compréhension des actions.

Les commentaires peuvent être placés sur une ligne à part ou bien à la fin d'une ligne contenant une commande.

Exemple :

```
# Ce programme affiche la date
date    # Cette ligne est la ligne qui affiche la date !
```

1.2. Variables

Comme dans tout langage de programmation, le script shell utilise des variables. Elles servent à stocker des informations en mémoire pour les réutiliser à volonté au cours du script.

Une variable est créée au moment où elle reçoit son contenu. Elle reste valide jusqu'à la fin de l'exécution du script ou à la demande explicite de l'auteur du script. Puisque le script est exécuté séquentiellement du début à la fin, il est impossible de faire appel à une variable avant qu'elle ne soit créée.

Le contenu d'une variable peut être modifié au cours du script, la variable continue d'exister. Si le contenu est supprimé, la variable reste active mais ne contient rien.

■ La notion de type de variable en script shell est possible mais est très peu utilisée. Le contenu d'une variable est toujours un caractère ou une chaîne de caractères.

```
Ê1 #!/usr/bin/env bash
Ê2
Ê3 #
Ê4 # Auteur : Antoine Le Morvan
Ê5 # Date : Janvier 2019
Ê6 # Version 1.0.0 : Sauvegarde dans /root les fichiers passwd, shadow, group et
gshadow
Ê7 #
Ê8
Ê9 # Variables globales
10 FICHER1=/etc/passwd
11 FICHER2=/etc/shadow
12 FICHER3=/etc/group
13 FICHER4=/etc/gshadow
14
15 # Dossier destination
16 DESTINATION=/root
17
18 # Nettoie l'Écran :
19 clear
20
21 # Lancer la sauvegarde
22 echo "La sauvegarde de $FICHER1, $FICHER2, $FICHER3, $FICHER4 vers $DESTINATION
va commencer : "
23
24 cp $FICHER1 $FICHER2 $FICHER3 $FICHER4 $DESTINATION
25
26 echo "La sauvegarde est terminÉe !"
```

Ce script fait usage de variables. Le nom d'une variable doit commencer par une lettre mais peut ensuite contenir n'importe quelle suite de lettres ou de chiffres. Hormis le tiret bas `_`, les caractères spéciaux ne sont pas utilisables.

Par convention, les variables créées par un utilisateur ont un nom en minuscules. Ce nom doit être choisi avec prudence pour ne pas être ni trop basique ni trop compliqué. Une variable peut toutefois être nommée avec des majuscules, comme c'est le cas ici, si il s'agit d'une variable globale qui ne doit pas être modifiée par le programme.

Le caractère `^` affecte du contenu à une variable :

```
variable=va leur
nom_rep="/home"
```

Il n'y a pas d'espace ni avant ni après le signe `=`.

Pour afficher du texte en même temps que le contenu d'une variable, il est obligatoire d'utiliser les guillemets et non les apostrophes.

L'usage des apostrophes inhibe l'interprétation des caractères spéciaux.

!

```
$ message="Bonjour"
$ echo "Voici le contenu de la variable message : $message"
Voici le contenu de la variable message : Bonjour
$ echo 'Voici le contenu de la variable message : $message'
Voici le contenu de la variable message : $message
```

Pour isoler le nom de la variable, il faut utiliser les apostrophes ou les accolades :

```
$ fichier=nom_fichier
$ touch "$fichier"1
$ touch ${fichier}1
```

!

L'utilisation systématique des accolades est conseillée.

Supprimer et verrouiller les variables

La commande `unset` permet de supprimer une variable.

Exemple :

```
$ nom="NOM"
$ prenom="Prenom"
$ echo "$nom $prenom"
NOM Prenom
$ unset prenom
$ echo "$nom $prenom"
NOM
```

La commande `readonly` ou `typeset -r` verrouille une variable.

Exemple :

```
$ nom="NOM"
$ readonly nom
$ nom="AUTRE NOM"
bash: nom: variable en lecture seule
$ unset nom
bash: nom: variable en lecture seule
```



Un `set -u` en d  but de script va arr  ter l  x  cution du script en cas d  utilisation des variables non d  clar  es.

Variables d  environnements

Les variables d  environnements et les variables syst  mes sont des variables utilis  es par le syst  me pour son fonctionnement. Par convention elles portent un nom en majuscules.

Comme toutes variables, elles peuvent   tre affich  es    l  x  cution d  un script. M  me si cela est fortement d  conseill  , elles peuvent aussi y   tre modifi  es.

   La commande `env` permet d  afficher toutes les variables d  environnement utilis  es.

   La commande `set` permet d  afficher toutes les variables syst  me utilis  es.

Parmi les dizaines de variables d  environnement, plusieurs ont un int  r  t      tre utilis  es dans un script shell :

Table 2. Variables d  environnement

Variable	Observation
<code>HOSTNAME</code>	Nom d��h��te de la machine.
<code>USER</code> , <code>USERNAME</code> et <code>LOGNAME</code>	Nom de l��utilisateur connect�� sur la session.
<code>PATH</code>	Chemin o�� trouver les commandes.
<code>PWD</code>	R��pertoire courant, mis �� jour �� chaque ex��cution de la commande <code>cd</code> .
<code>HOME</code>	R��pertoire de connexion.
<code>\$\$</code>	Num��ro du processus de l��x��cution du script.
<code>\$?</code>	Code retour de la derni��re commande ex��cut��e.

Exporter une variable

La commande `export` permet d  exporter une variable.

Une variable n  est valable que dans l  environnement du processus du script shell. Pour que les processus fils du script puissent conna  tre les variables et leurs contenus, il faut les exporter.

La modification d  une variable export  e dans un processus fils ne peut pas remonter au processus p  re.



Sans option, la commande `export` affiche le nom et les valeurs des variables export  es dans l  environnement.

La substitution de commande

Il est possible de stocker le résultat d'une commande dans une variable.

!

Cette opération n'est valable que pour les commandes qui renvoient un message à la fin de leur exécution.

La syntaxe pour sous-exécuter une commande est la suivante :

Syntaxes pour la substitution de commandes

```
variable=`commande`  
variable=$(commande) # Syntaxe à privilégier
```

Exemple :

```
$ jour=`date +%j`  
$ homedir=$(pwd)
```

Améliorations du script de sauvegarde

Quelques pistes d'améliorations

```
#!/usr/bin/env bash  
#  
#  
# Auteur : Antoine Le Morvan  
# Date : Janvier 2019  
# Version 1.0.0 : Sauvegarde dans /root les fichiers passwd, shadow, group et  
# gshadow  
# Version 1.0.1 : Création d'un répertoire avec le quantième du jour.  
# Améliorations diverses  
#  
# Variables globales  
## Fichiers à sauvegarder  
FICHER1=/etc/passwd  
FICHER2=/etc/shadow  
FICHER3=/etc/group  
FICHER4=/etc/gshadow  
## Dossier destination  
DESTINATION=/root  
## Variables en readonly  
readonly FICHER1  
readonly FICHER2
```



```

24 readonly FICHIER3
25 readonly FICHIER4
26 readonly DESTINATION
27
28 # Un nom de dossier avec le quantième du jour
29 rep="backup-$(date +%j)"
30
31 # Nettoie l'Écran :
32 clear
33
34 # Lancer la sauvegarde
35 echo "*****"
36 echo "      Script de sauvegarde - Sauvegarde sur la machine $HOSTNAME "
37 echo "*****"
38 echo "La sauvegarde sera faite dans le dossier ${rep}."
39 echo "CrÉation du rÉpertoire..."
40 mkdir -p $DESTINATION/$rep
41 echo "                                     [ OK ]"
42 echo "La sauvegarde de ${FICHIER1}, ${FICHIER2}, ${FICHIER3}, ${FICHIER4} vers
${DESTINATION}/${rep} va commencer :"
43
44 cp $FICHIER1 $FICHIER2 $FICHIER3 $FICHIER4 $DESTINATION/$rep
45
46 echo "La sauvegarde est terminÉe !"
47
48 # La sauvegarde est notÉe dans le journal d'ÉvÉnements du systÉme :
49 echo "La sauvegarde est renseignÉe dans syslog :"
50 logger "Sauvegarde des fichiers systÉmes par ${USER} sur la machine ${HOSTNAME}
dans le dossier ${DESTINATION}/${rep}."
51 echo "                                     [ OK ]"

```

ExÉcution de notre script de sauvegarde

```

root # ./02-backup-enhanced.sh
*****
É      Script de sauvegarde - Sauvegarde sur la machine formateur1
*****
La sauvegarde sera faite dans le dossier backup-262.
CrÉation du rÉpertoire...
É                                     [ OK ]
La sauvegarde de /etc/passwd, /etc/shadow, /etc/group, /etc/gshadow vers /root/backup-
262 va commencer :
La sauvegarde est terminÉe !
La sauvegarde est renseignÉe dans syslog :
É                                     [ OK ]

```

Grace à la commande `logger`, les affichages de l'exÉcution du script sont Écrites dans le journal `syslog` :

```
root # tail -f /var/log/messages
janvier. 02 19:35:35 formateur1 antoine[9712]: Sauvegarde des fichiers syst•mes par
antoine sur la machine formateur1 dans le dossier /root/b...
```

1.3. Saisie et manipulations

Selon l’objet du script, il peut •tre nŽcessaire de lui envoyer des informations lors de son lancement ou durant son exŽcution.

Ces informations, non connues lors de l’Žcriture du script, peuvent •tre extraites • partir de fichiers ou saisies par l’utilisateur.

Il est aussi possible d’envoyer ces informations sous forme d’arguments lors de la saisie de la commande du script. C’est le mode de fonctionnement de nombreuses commandes Linux.

La commande read

La commande `read` permet de saisir une cha•ne de caract•res pour la stocker dans une variable.

Syntaxe de la commande read

```
read [-n X] [-p] [-s] [variable]
```

Exemple de la commande read

```
$ read nom prenom
$ read -p "Veuillez saisir votre nom : " nom
```

Table 3. Options de la commande read

Option	Observation
-p	Affiche un message de prompt
-n	Limite le nombre de caract•res • saisir
-s	Masque la saisie

Lors de l’utilisation de l’option `-n`, le shell valide automatiquement la saisie au bout du nombre de caract•res prŽcisŽs. L’utilisateur n’a pas • appuyer sur la touche `[ENTREE]`.

```
$ read -n5 nom
```

La commande `read` permet d’interrompre l’exŽcution du script le temps que l’utilisateur saisisse des informations. La saisie de l’utilisateur est dŽcoupŽe en mots affectŽs • une ou plusieurs variables

préfinies. Les mots sont des chaînes de caractères séparées par le séparateur de champs.

La fin de la saisie est déterminée par la frappe sur la touche **[ENTREE]** ou le caractère spécial de fin de ligne.

Une fois la saisie validée, chaque mot sera stocké dans la variable préfinie.

Le découpage des mots est défini par le caractère séparateur de champs. Ce séparateur est stocké dans la variable système **IFS** (*Internal Field Separator*).

```
$ set | grep IFS
IFS=$' \t\n'
```

Par défaut, IFS contient l'espace, la tabulation **\t** et le saut de ligne **\n**.

Utilisée sans préciser de variable, cette commande met simplement le script en pause. Le script continue son exécution lors de la validation de la saisie.

Cette utilisation permet de faire une pause lors du débogage d'un script ou pour inciter l'utilisateur à appuyer sur **ENTREE** pour continuer.

```
$ echo -n "Appuyer sur [ENTREE] pour continuer..."
$ read
```

La commande cut

La commande **cut** permet d'isoler une colonne dans un fichier ou dans un flux.

Syntaxe de la commande cut

```
cut [-cx] [-dy] [-fz] fichier
```

Exemple d'utilisation de la commande cut

```
$ cut -d: -f1 /etc/passwd
```

Table 4. Options de la commande cut

Option	Observation
-c	Spécifie les numéros d'ordre des caractères à sélectionner
-d	Spécifie le séparateur de champs
-f	Spécifie le numéro d'ordre des colonnes à sélectionner

Le principal intérêt de cette commande sera son association avec un flux, par exemple la

commande **grep** et le pipe **|**.

¥ La commande **grep** travaille verticalement (*isolation d'une ligne parmi toutes celles du fichier*).

¥ La combinaison des deux commandes permet d'isoler un champ précis du fichier.

Syntaxe de la commande **cut**

```
# grep "^root:" /etc/passwd | cut -d: -f3
0
```

!

Les fichiers de configurations comportant une structure unique utilisant le même séparateur de champs sont des cibles idéales pour cette combinaison de commandes.

La commande **tr**

La commande **tr** permet de convertir une chaîne de caractères.

Syntaxe de la commande **tr**

```
tr [-csd] chaîne1 chaîne2
```

Table 5. Options de la commande **tr**

Option	Observation
-c	Tous les caractères qui ne sont pas spécifiés dans la première chaîne sont convertis selon les caractères de la seconde.
-d	Efface le caractère spécifié.
-s	Réduire ^ une seule unité le caractère spécifié.

Exemple d'utilisation de la commande **tr**

```
$ grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
$ grep root /etc/passwd | tr -s "o"
rot:x:0:0:rot:/rot:/bin/bash
```

Exercice : Extraire le niveau d'exécution du fichier **/etc/inittab**

```

Ê1 #!/usr/bin/env bash
Ê2
Ê3 #
Ê4 # Auteur : Antoine Le Morvan
Ê5 # Date : Janvier 2019
Ê6 # Version 1.0.0 : Extrait le niveau d'exécution du fichier /etc/inittab
Ê7
Ê8 # Variables globales
Ê9
10 INITTAB=/etc/inittab
11
12 niveau=`grep "^id" $INITTAB | cut -d: -f2`
13
14 # Affichage du résultat :
15 echo "Le niveau d'init au démarrage est : $niveau"

```

Extraire le nom et le chemin d'un fichier

¥ La commande `basename` permet d'extraire le nom du fichier à partir d'un chemin.

¥ La commande `dirname` permet d'extraire le chemin parent d'un fichier.

Exemple :

```

$ echo $FICHIER=/usr/bin/passwd
$ basename $FICHIER
passwd
$ dirname $FICHIER
/usr/bin

```

Arguments d'un script

La demande de saisie d'informations grâce à la commande `read` interrompt l'exécution du script tant que l'utilisateur ne fait pas de saisie.

Cette méthode, bien que très conviviale, présente des limites si l'on agit d'un script à l'exécution programmée la nuit par exemple. Afin de palier ce problème, il est possible d'injecter les informations souhaitées via des arguments.

De nombreuses commandes Linux fonctionnent sur ce principe.

Cette façon de faire a l'avantage qu'une fois le script exécuté, il n'aura plus besoin d'intervention humaine pour se terminer.

Son inconvénient majeur est qu'il faudra prévenir l'utilisateur de la syntaxe du script pour éviter des erreurs.

Les arguments sont renseignés lors de la saisie de la commande du script. Ils sont séparés par un espace.

```
$ ./script argument1 argument2
```

Une fois exécuté, le script enregistre les arguments saisis dans des variables prédéfinies : les variables positionnelles.

Ces variables sont utilisables dans le script comme n'importe quelle autre variable, à l'exception faite qu'elles ne peuvent pas être affectées.

¥ Les variables positionnelles non utilisées existent mais sont vides.

¥ Les variables positionnelles sont toujours définies de la même façon :

Table 6. Les variables positionnelles

Variable	Observation
<code>\$0</code>	contient le nom du script tel qu'il a été saisi.
<code>\$1 ^ \$9</code>	contiennent les valeurs du 1er et du 9 ^{ème} argument.
<code>\${x}</code>	contient la valeur de l'argument <code>x</code> , supérieur ^ 9.
<code>\$#</code>	contient le nombre d'arguments passés.
<code>\$* ou \$@</code>	contient en une variable tous les arguments passés

Exemples :

```
Ê1 #!/usr/bin/env bash
Ê2 #
Ê3 # Auteur : Damien dit LeDub
Ê4 # Date : septembre 2019
Ê5 # Version 1.0.0 : Affiche la valeur des arguments positionnels
Ê6 # de 1 ^ 3
Ê7
Ê8 # Le sŽparateur de champ sera ", " ou l'espace
Ê9 # Important pour visualiser la diffŽrence en $* et $@
10 IFS=", "
11
12 # Affiche un texte ^ l'Žcran :
13 echo "Le nombre d'argument (\$#) = $#\"
14 echo "Le nom du script      (\$0) = $0\"
15 echo "Le 1er argument      (\$1) = $1\"
16 echo "Le 2e argument       (\$2) = $2\"
17 echo "Le 3e argument       (\$3) = $3\"
18 echo "Tous sŽparŽs par IFS (\$*) = $*\"
19 echo "Tous sans sŽparation (\$@) = $@\"
```

```
$ ./un_deux_trois.sh un deux "trois quatre"
Le nombre d'argument ($#) = 3
Le nom du script      ($0) = ./un_deux_trois.sh
Le 1er argument       ($1) = un
Le 2e argument        ($2) = deux
Le 3e argument        ($3) = trois quatre
Tous sŽparŽs par IFS ($*) = un,deux,trois quatre
Tous sans sŽparation ($@) = un deux trois quatre
```

Attention ^ la diffŽrence entre \$@ et \$*.

Elle se fait au niveau du format de stockage des arguments :

¥ \$* : Contient les arguments au format "\$1 \$2 \$3 É "

¥ \$@ : Contient les arguments au format "\$1" "\$2" "\$3" É

Cœst en modifiant la variable dœnvironnement IFS que la diffŽrence est visible.

La commande shi ft

La commande shi ft permet de dŽcaler les variables positionnelles.

Exemples :

```
Ê1 #!/usr/bin/env bash
Ê2 #
Ê3 # Auteur : Damien dit LeDub
Ê4 # Date : septembre 2019
Ê5 # Version 1.0.0 : Affiche la valeur des arguments positionnels
Ê6 # de 1 ^ 3 avant et après un ÇshiftÊ
Ê7
Ê8 # Le sŽparateur de champ sera ",", " ou l'espace
Ê9 # Important pour visualiser la diffŽrence en $* et @$
10 IFS=", "
11
12 # RŽcup•re les arguments avant le ÇshiftÊ
13 AVANT_SHIFT_NB="$#"
14 AVANT_SHIFT_NOM="$0"
15 AVANT_SHIFT_1ER="$1"
16 AVANT_SHIFT_2E="$2"
17 AVANT_SHIFT_3E="$3"
18 AVANT_SHIFT_TOUS_IFS="$*"
19 AVANT_SHIFT_TOUS="$@"
20
21 # RŽcup•re les arguments après le ÇshiftÊ
22 shift 2
23 APRES_SHIFT_NB="$#"
24 APRES_SHIFT_NOM="$0"
25 APRES_SHIFT_1ER="$1"
26 APRES_SHIFT_2E="$2"
27 APRES_SHIFT_3E="$3"
28 APRES_SHIFT_TOUS_IFS="$*"
29 APRES_SHIFT_TOUS="$@"
30
31 # Affiche toutes les informations rŽcupŽrŽes
32 echo "Le nombre d'argument (\$#) : ${AVANT_SHIFT_NB}"
33 echo "  après le shift 2      : ${APRES_SHIFT_NB}"
34 echo "Le nom du script      (\$0) : ${AVANT_SHIFT_NOM}"
35 echo "  après le shift 2      : ${APRES_SHIFT_NOM}"
36 echo "Le 1er argument        (\$1) : ${AVANT_SHIFT_1ER}"
37 echo "  après le shift 2      : ${APRES_SHIFT_1ER}"
38 echo "Le 2e argument          (\$2) : ${AVANT_SHIFT_2E}"
39 echo "  après le shift 2      : ${APRES_SHIFT_2E}"
40 echo "Le 3e argument          (\$3) : ${AVANT_SHIFT_3E}"
41 echo "  après le shift 2      : ${APRES_SHIFT_3E}"
42 echo "Tous sŽparŽ par IFS  (\$*) : ${AVANT_SHIFT_TOUS_IFS}"
43 echo "  après le shift 2      : ${APRES_SHIFT_TOUS_IFS}"
44 echo "Tous sans sŽparation (\$@) : ${AVANT_SHIFT_TOUS}"
45 echo "  après le shift 2      : ${APRES_SHIFT_TOUS} "
```



```

$ ./un_deux_trois_shift.sh un deux "trois quatre"
Le nombre d'argument ($#) : 3
Ê après le shift 2      : 1
Le nom du script ($0) : ./un_deux_trois_shift.sh
Ê après le shift 2      : ./un_deux_trois_shift.sh
Le 1er argument ($1) : un
Ê après le shift 2      : trois quatre
Le 2e argument ($2) : deux
Ê après le shift 2      :
Le 3e argument ($3) : trois quatre
Ê après le shift 2      :
Tous sŽparŽ par IFS ($*) : un,deux,trois quatre
Ê après le shift 2      : trois quatre
Tous sans sŽparation ($@) : un deux trois quatre
Ê après le shift 2      : trois quatre

```

- Lors de l'utilisation de la commande `shift`, les variables `$#` et `$*` sont modifiŽes en consŽquence.

La commande `set`

La commande `set` dŽcoupe une chaŽne en variables positionnelles.

Syntaxe de la commande `set`

```
set [valeur] [$variable]
```

Exemple :

```

$ set un deux trois
$ echo $1 $2 $3 $#
un deux trois 3
$ variable="un deux trois"
$ set $variable
$ echo $1 $2 $3 $#
un deux trois 3

```

Ci-dessous, la version de notre script de sauvegarde mettant en oeuvre les variables positionnelles :

```

Ê1 #!/usr/bin/env bash
Ê2
Ê3 #
Ê4 # Auteur : Antoine Le Morvan
Ê5 # Date : Janvier 2019
Ê6 # Version 1.0.0 : Sauvegarde dans /root les fichiers passwd, shadow, group et
gshadow
Ê7 # Version 1.0.1 : Cr ation d'un r pertoire avec le quanti me du jour.
Ê8 #             Am liorations diverses
Ê9 # Version 1.0.2 : Modification pour utiliser les variables positionnelles
10 #             Limitation ^ 5 fichiers
11
12 # Variables globales
13
14 ## Dossier destination
15 DESTINATION=/root
16
17 # Un nom de dossier avec le quantieme du jour
18 rep="backup-$(date +%j)"
19
20 # Nettoie l' cran :
21 clear
22
23 # Lancer la sauvegarde
24 echo "*****"
25 echo "      Script de sauvegarde - Sauvegarde sur la machine $HOSTNAME "
26 echo "*****"
27 echo "La sauvegarde sera fa te dans le dossier ${rep}."
28 echo "Cr ation du r pertoire..."
29 mkdir -p $DESTINATION/$rep
30 echo "                                [ OK ]"
31 echo "La sauvegarde de ${1} ${2} ${3} ${4} ${5} vers ${DESTINATION}/${rep} va
commencer : "
32
33 cp $1 $2 $3 $4 $5 $DESTINATION/$rep
34
35 echo "La sauvegarde est termin e !"

```

Tester vos connaissances

Parmi ces 4 shells, lequel n existe pas :

- " Bash
- " Ksh
- " Tsh
- " Csh

```
# Quelle est la bonne syntaxe pour affecter un contenu à une variable :
" variable:=valeur
" variable := valeur
" variable = valeur
" variable=valeur

# Comment stocker le retour d'une commande dans une variable :
" fichier=$(ls)
" fichier=`ls`
" fichier:=$ls
" fichier = $(ls)

# La commande read permet de lire le contenu d'un fichier :
" Vrai
" Faux

# Parmi les propositions ci-dessous, laquelle est la bonne syntaxe pour la commande cut :
" cut -f: -d1 /etc/passwd
" cut -d: -f1 /etc/passwd
" cut -d1 -f: /etc/passwd
" cut -c ":" -f 3 /etc/passwd

# Quelle commande permet de décaler des variables positionnelles :
" left
" shift
" set
" declare

# Quelle commande transforme une chaîne de caractères en variables positionnelles :
" left
" shift
" set
" array
```

Chapitre 2. Les scripts shell - Instructions de contrôle

" Objectifs pédagogiques

Dans ce chapitre, vous allez apprendre à utiliser des structures de tests, des structures conditionnelles et des boucles.

- # Tester des variables, des fichiers ;
- # Utiliser une structure conditionnelle ;
- # Faire des boucles `while`, `until`, `select`, `for`.

\$ script, shell, bash, structure, boucle

Connaissances : % %

Niveau technique : ! !

Temps de lecture : 20 minutes

Lorsqu'elles se terminent, toutes les commandes exécutées par le shell renvoient un code de retour (également appelé code de statut ou de sortie).

2.1. Les tests

- ¥ Si la commande s'est correctement exécutée, la convention veut que le code de statut ait pour valeur zéro.
- ¥ Si la commande a rencontré un problème lors de son exécution, son code de statut aura une valeur différente de zéro.
Les raisons peuvent être nombreuses : manque de droits d'accès, absence de fichier, saisie incorrecte, etc.

Il faut se référer au manuel de la commande `man commande` pour connaître les différentes valeurs du code de retour prévues par les développeurs.

Le code de retour n'est pas visible directement, mais est enregistré dans une variable spéciale : `$?`.

```
$ mkdir repertoire
$ echo $?
0
$ mkdir /repertoire
mkdir: impossible de cr  er le r  pertoire
$ echo $?
1
$ commande_qui_n_existe_pas
commande_qui_n_existe_pas  : commande introuvable
$ echo $?
127
```

!

L'affichage du contenu de la variable `$?` avec la commande `echo` se fait imm  diatement apr  s la commande que l'on souhaite   valuer car cette variable est mise    jour apr  s chaque ex  cution d'une commande, d'une ligne de commandes ou encore d'un script.

Puisque la valeur de `$?` change apr  s chaque ex  cution de commande, il est pr  f  rable de mettre sa valeur dans une variable qui sera utilis  e par la suite, pour un test ou afficher un message.

#

```
$ ls fichier_absent
ls: impossible d'acc  der    'fichier_absent': Aucun fichier ou dossier
de ce type
$ RETOUR=$?
$ echo $?
0
$ echo $RETOUR
2
```

Il est   galement possible de cr  er des codes de retour dans un script. Il suffit pour cela d'ajouter un argument num  rique    la commande `exit`.

```
$ bash          # pour   viter d'  tre d  connect   apr  s le   xit 2   
$ exit 2
$ echo $?
2
```

Outre la bonne ex  cution d'une commande, le shell offre la possibilit   d'ex  cuter des tests sur de nombreux motifs :

-    Fichiers : existence, type, droits, comparaison ;
-    Cha  nes de caract  res : longueur, comparaison ;

¥ NumŽriques entiers : valeur, comparaison.

Le rŽsultat du test :

¥ \$?=0 : le test s’est correctement exŽcutŽ et est vrai ;

¥ \$?=1 : le test s’est correctement exŽcutŽ et est faux ;

¥ \$?=2 : le test ne s’est pas correctement exŽcutŽ.

Tester le type d’un fichier

Syntaxe de la commande test pour un fichier

```
test [-d|-e|-f|-L] fi chi er
```

Table 7. Options de la commande test sur les fichiers

Option	Observation
-e	Teste si le fichier existe
-f	Teste si le fichier existe et est de type normal
-d	Teste si le fichier existe et est de type rŽpertoire
-L	Teste si le fichier existe et est de type lien symbolique
-b	Teste si le fichier existe et est de type spŽcial mode bloc
-c	Teste si le fichier existe et est de type spŽcial mode caract•re
-p	Teste si le fichier existe et est de type tube
-S	Teste si le fichier existe et est de type socket
-t	Teste si le fichier existe et est de type terminal
-r	Teste si le fichier existe et est accessible en lecture
-w	Teste si le fichier existe et est accessible en Žcriture
-x	Teste si le fichier existe et est est exŽcutable
-g	Teste si le fichier existe et est a un SGID positionnŽ
-u	Teste si le fichier existe et est a un SUID positionnŽ
-s	Teste si le fichier existe et est non vide (taille > 0 octets)

Comparer deux fichiers

La commande test peut Žgalement comparer des fichiers :

Syntaxe de la commande test pour la comparaison de fichiers

```
test fi chi er1 [-nt|-ot|-ef] fi chi er2
```

Table 8. Options de la commande test pour la comparaison de fichiers

Option	Observation
-nt	Teste si le premier fichier est plus récent que le second
-ot	Teste si le premier fichier est plus ancien que le second
-ef	Teste si le premier fichier est un lien physique du second

Tester une variable

Syntaxe de la commande test pour les variables

```
test [-z|-n] $variable
```

Table 9. Options de la commande test pour les variables

Option	Observation
-z	Teste si la variable est vide
-n	Teste si la variable n'est pas vide

Tester une chaîne de caractères

Syntaxe de la commande test pour les chaînes de caractères

```
test chaîne1 [=|!=] chaîne2
```

Exemple :

```
$ test "$var" = "Hello world !"
$ echo $?
0
```

Table 10. Options de la commande test pour les variables

Option	Observation
=	Teste si la première chaîne de caractères est égale à la seconde
!=	Teste si la première chaîne de caractères est différente de la seconde
<	Teste si la première chaîne de caractères est avant la seconde dans l'ordre ASCII
>	Teste si la première chaîne de caractères est après la seconde dans l'ordre ASCII

Comparaison de numŽriques entiers

Syntaxe de la commande test pour les entiers

```
test "num1" [-eq|-ne|-gt|-lt] "num2"
```

Exemple :

```
$ var=1
$ test "$var" -eq "1"
$ echo $?
0
$ var=2
$ test "$var" -eq "1"
$ echo $?
1
```

Table 11. Options de la commande test pour les entiers

Option	Observation
-eq	Teste si le premier nombre est Žgal au second
-ne	Teste si le premier nombre est diffŽrent au second
-gt	Teste si le premier nombre est supŽrieur au second
-lt	Teste si le premier nombre est infŽrieur au second

Les numŽriques Žtant traitŽs par le shell comme des caract•res (ou cha”nes de caract•res) classiques, un test sur un caract•re peut renvoyer le m•me rŽsultat qu’il soit traitŽ en tant que numŽrique ou non.

!

```
$ test "1" = "1"
$ echo $?
0
$ test "1" -eq "1"
$ echo $?
0
```

Mais le rŽsultat du test n’aura pas la m•me signification :

- ¥ Dans le premier cas, il signifiera que les deux caract•res ont la m•me valeur dans la table ASCII.
- ¥ Dans le second cas, il signifiera que les deux nombres sont Žgaux.

Combinaison de tests

La combinaison de tests permet d'effectuer plusieurs tests en une seule commande. Il est possible de tester plusieurs fois le même argument (fichier, chaîne ou numérique) ou des arguments différents.

```
test option1 argument1 [-a|-o] option2 argument 2
```

```
$ ls -lad /etc
drwxr-xr-x 142 root root 12288 sept. 20 09:25 /etc
$ test -d /etc -a -x /etc
$ echo $?
0
```

Table 12. Options de combinaison de tests

Option	Observation
-a	ET : Le test sera vrai si tous les motifs le sont.
-o	OU : Le test sera vrai si au moins un motif l'est.

Les tests peuvent ainsi être groupés avec des parenthèses () pour leur donner une priorité.

```
(TEST1 -a TEST2) -a TEST3
```

Le caractère ! permet d'effectuer le test inverse de celui demandé par l'option :

```
$ test -e /fichier # vrai si fichier existe
$ ! test -e /fichier # vrai si fichier n'existe pas
```

Les opérations numériques

La commande `expr` effectue une opération avec des entiers numériques.

Syntaxe de la commande expr

```
expr num1 [+] [-] [\*] [/] [%] num2
```

Exemple :

Exemple d'utilisation de la commande `expr`

```
$ expr 2 + 2
4
```



Attention ^ bien encadrer le signe d'opérateur par une espace, vous obtiendrez un message d'erreur en cas d'oubli.

Dans le cas d'une multiplication, le caractère joker `*` est précédé par `\` pour éviter une mauvaise interprétation.

Table 13. Opérateurs de la commande `expr`

Opérateur	Observation
<code>+</code>	Addition
<code>-</code>	Soustraction
<code>*</code>	Multiplication
<code>/</code>	Quotient de la division
<code>%</code>	Modulo de la division

La commande `typeset`

La commande `typeset -i` déclare une variable comme un entier.

Exemple :

Exemple d'utilisation de la commande `typeset`

```
$ typeset -i var1
$ var1=1+1
$ var2=1+1
$ echo $var1
2
$ echo $var2
1+1
```

La commande `let`

La commande `let` teste si un caractère est numérique.

Exemple :

Exemple d'utilisation de la commande `let`

```
$ var1="10"
$ var2="AA"
$ let $var1
$ echo $?
0
$ let $var2
$ echo $?
1
```

La commande `let` ne retourne pas un code retour cohérent lorsqu'elle value le numérique 0.

&

```
$ let 0
$ echo $?
1
```

La commande `let` permet également d'effectuer des opérations mathématiques :

```
$ let var=5+5
$ echo $var
10
```

`let` peut être substitué par `$(())`.

!

```
$ echo $((5+2))
7
$ echo $((5*2))
10
$ var=$((5*3))
$ echo $var
15
```

2.2. Structures conditionnelles

Si la variable `$?` permet de connaître le résultat d'un test ou de l'exécution d'une commande, elle ne peut être affichée et n'a aucune incidence sur le déroulement d'un script.

Mais nous pouvons nous en servir dans une condition. Si le test est bon alors je fais cette action sinon je fais telle autre action.

Syntaxe de l'alternative conditionnelle `if`

```
if commande
then
  É commande si $?=0
else
  É commande si $?!=0
fi
```

La commande placée après le mot `if` peut être n'importe quelle commande puisque c'est son code de retour (`?`) qui sera évalué. Il est souvent pratique d'utiliser la commande `test` pour définir plusieurs actions en fonction du résultat de ce test (fichier existe, variable non vide, droits en écriture positionnés).

Utiliser une commande classique (`mkdir`, `tar`, `É`) permet de définir les actions à effectuer en cas de succès ou les messages d'erreur à afficher en cas d'échec.

Exemples d'utilisation de la structure conditionnelle `if`

```
if test -e /etc/passwd
then
  É echo "Le fichier existe"
else
  É echo "Le fichier n'existe pas"
fi

if mkdir rep
then
  É cd rep
fi
```

La commande `test` peut être substituée par `[[condition_de_test]]`.

Ainsi :

#

```
if test -e /etc/passwd
then
    echo "Le fichier existe"
else
    echo "Le fichier n'existe pas"
fi
```

peut devenir :

```
if [[ -e /etc/passwd ]]
then
    echo "Le fichier existe"
else
    echo "Le fichier n'existe pas"
fi
```

Si le bloc `else` commence par une nouvelle structure `if`, il est possible de fusionner `else` et `if` :

```
[É]
else
    if test -e /etc/
[É]

[É]
# est équivalent à
elif test -e /etc
[É]
```

La structure `if / then / else / fi` évalue la commande placée après `if` :

¥ Si le code retour de cette commande est 0 (vrai) le shell exécutera les commandes placées après `then` ;

¥ Si le code retour est différent de 0 (faux) le shell exécutera les commandes placées après `else`.

Le bloc `else` est facultatif.

Il existe un besoin d'effectuer certaines actions uniquement si l'évaluation de la commande est vraie et n'avoir rien à faire si elle est fausse.

Le mot `fi` ferme la structure.

Lorsqu'il n'y a qu'une seule commande à exécuter dans le bloc `then`, il est possible d'utiliser une syntaxe plus simple.

La commande à exécuter si `$?` est vrai est placée après `&&` tandis que la commande à exécuter si `$?` est faux est placée après `||` (facultatif).

Par exemple :

```
$ test -e /etc/passwd && echo "Le fichier existe" || echo "Le fichier n'existe pas"
$ mkdir repert && echo "Le répertoire est créé"
```

Il est possible d'évaluer et de remplacer une variable avec une structure plus légère que `if`.

Cette syntaxe met en œuvre les accolades :

¥ Affiche une valeur de remplacement si la variable est vide :

```
${variable:-valeur}
```

¥ Affiche une valeur de remplacement si la variable n'est pas vide :

```
${variable:+valeur}
```

¥ Affecte une nouvelle valeur à la variable si elle est vide :

```
${variable:=valeur}
```

Exemples :

```
$ nom=""
$ echo ${nom:-linux}
linux
$ echo $nom

$ echo ${nom:=linux}
linux
$ echo $nom
linux
$ echo ${nom:+tux}
tux
$ echo $nom
linux
```

Structure alternative conditionnelle case

Une succession de structures `if` peut vite devenir lourde et complexe. Lorsqu'elle concerne l'évaluation d'une même variable, il est possible d'utiliser une structure conditionnelle à plusieurs branches. Les valeurs de la variable peuvent être précisées ou appartenir à une liste de possibilités.

Les caractères jokers sont utilisables.

La structure `case` est `in / esac` évalue la variable placée après `case` et la compare aux valeurs définies.

À la première égalité trouvée, les commandes placées entre `)` et `;;` sont exécutées.

La variable évaluée et les valeurs proposées peuvent être des chaînes de caractères ou des résultats de sous-exécutions de commandes.

Placé en fin de structure, le choix `*` indique les actions à exécuter pour toutes les valeurs qui n'ont pas été précédemment testées.

Syntaxe de l'alternative conditionnelle case

```
case $variable in
  valeur1)
    commandes si $variable = valeur1
    ;;
  valeur2)
    commandes si $variable = valeur2
    ;;
  [...]
  *)
    commandes pour toutes les valeurs de $variable != de valeur1 et valeur2
    ;;
esac
```

Lorsque la valeur est sujette à variation, il est conseillé d'utiliser les caractères jokers `[]` pour spécifier les possibilités :

```
[Oo][Uu][Ii])
  echo "oui"
;;
```

Le caractère `|` permet aussi de spécifier une valeur ou une autre :

```
"oui" | "OUI")
  echo "oui"
;;
```

2.3. Boucles

Le shell bash permet l'utilisation de boucles. Ces structures permettent l'exécution d'un bloc de commandes plusieurs fois (de 0 à l'infini) selon une valeur définie statiquement, dynamiquement ou sur condition :

```
while  
until  
for  
select
```

Quelle que soit la boucle utilisée, les commandes à répéter se placent entre les mots `do` et `done`.

La structure boucle conditionnelle `while`

La structure `while / do / done` évalue la commande placée après `while`.

Si cette commande est vraie (`$? = 0`), les commandes placées entre `do` et `done` sont exécutées. Le script retourne ensuite au début à évaluer de nouveau la commande.

Lorsque la commande évaluée est fautive (`$? != 0`), le shell reprend l'exécution du script à la première commande après `done`.

Syntaxe de la structure boucle conditionnelle `while`

```
while commande  
do  
  É commande si $? = 0  
done
```

Exemple :

Exemple d'utilisation de la structure conditionnelle `while`

```
while test -e /etc/passwd  
do  
  É echo "Le fichier existe"  
done
```

&

Si la commande évaluée ne varie pas, la boucle sera infinie et le shell n'exécutera jamais les commandes placées à la suite du script. Cela peut être volontaire, mais aussi être une erreur.

Il faut donc faire très attention à la commande qui régule la boucle et trouver un moyen d'en sortir.

Pour sortir d'une boucle `while`, il faut faire en sorte que la commande `while` ne soit plus vraie, ce qui n'est pas toujours possible.

Il existe des commandes qui permettent de modifier le comportement d'une boucle :

• `exit`

• `break`

• `continue`

La commande `exit`

La commande `exit` termine l'exécution du script.

Syntaxe de la commande `exit`

```
exit [n]
```

Exemple :

Exemple d'utilisation de la commande `exit`

```
$ bash          # pour éviter d'être déconnecté après le C exit 1 È
$ exit 1
$ echo $?
1
```

La commande `exit` met fin au script immédiatement. Il est possible de préciser le code de retour du script en le précisant en argument (de 0 à 255). Sans argument précisé, c'est le code de retour de la dernière commande du script qui sera transmise à la variable `?`.

Cette commande est utile dans le cas d'un menu proposant la sortie du script dans les choix possibles.

La commande `break` / `continue`

La commande `break` permet d'interrompre la boucle en allant à la première commande après `done`.

La commande `continue` permet de relancer la boucle en revenant à la première commande après `do`.

```
while test -d /
do
  echo "Voulez-vous continuer ? (oui/non)"
  read rep
  test $rep = "oui" && continue
  test $rep = "non" && break
done
```

Les commandes true / false

La commande `true` renvoie toujours vrai tandis que la commande `false` renvoie toujours faux.

```
$ true
$ echo $?
0
$ false
$ echo $?
1
```

Utilisées comme condition d'une boucle, elles permettent soit d'exécuter une boucle infinie soit de désactiver cette boucle.

Exemple :

```
while true
do
  echo "Voulez-vous continuer ? (oui/non)"
  read rep
  test $rep = "oui" && continue
  test $rep = "non" && break
done
```

La structure boucle conditionnelle until

La structure `until` / `do` / `done` évalue la commande placée après `until`.

Si cette commande est fausse (`$? != 0`), les commandes placées entre `do` et `done` sont exécutées. Le script retourne ensuite au début à évaluer de nouveau la commande.

Lorsque la commande évaluée est vraie (`$? = 0`), le shell reprend l'exécution du script à la première commande après `done`.

Syntaxe de la structure boucle conditionnelle until

```
until commande
do
  Ê commande si $? != 0
done
```

Exemple :

Exemple d'utilisation de la structure conditionnelle until

```
until test -e /etc/passwd
do
  Ê echo "Le fichier n'existe pas"
done
```

La structure choix alternatif select

La structure `select / do / done` permet d'afficher rapidement un menu avec plusieurs choix et une demande de saisie.

À chaque `select` de la liste correspond un choix numéroté. À la saisie, la valeur choisie est affectée à la variable placée après `select` (créée à cette occasion).

Elle exécute ensuite les commandes placées entre `do` et `done` avec cette valeur.

¥ La variable `PS3` contient le message d'invitation à entrer le choix ;

¥ La variable `REPLY` va permettre de récupérer le numéro du choix.

Il faut une commande `break` pour sortir de la boucle.

!

La structure `select` est très utile pour de petits menus simples et rapides. Pour personnaliser un affichage plus complet, il faudra utiliser les commandes `echo` et `read` dans une boucle `while`.

Syntaxe de la structure boucle conditionnelle select

```
PS3="Votre choix : "
select variable in var1 var2 var3
do
  Ê commandes
done
```

Exemple :

Exemple d'utilisation de la structure conditionnelle select

```
PS3="Votre choix : "  
select choix in café thé chocolat  
do  
  Écho "Vous avez choisi le $REPLY : $choix"  
done
```

ce qui donne l'exécution :

```
1) Café  
2) Thé  
3) Chocolat  
Votre choix : 2  
Vous avez choisi le choix 2 : thé  
Votre choix :
```

La structure boucle sur liste de valeurs for

La structure **for** / **do** / **done** affecte le premier élément de la liste à la variable placée après **for** (créée à cette occasion).

Elle exécute ensuite les commandes placées entre **do** et **done** avec cette valeur. Le script retourne ensuite au début affecter l'élément suivant de la liste à la variable de travail.

Lorsque le dernier élément a été utilisé, le shell reprend l'exécution à la première commande après **done**.

Syntaxe de la structure boucle sur liste de valeurs for

```
for variable in liste  
do  
  commandes  
done
```

Exemple :

Exemple d'utilisation de la structure conditionnelle for

```
for fichier in /home /etc/passwd /root/fic.txt  
do  
  Écho "Fichier $fichier"  
done
```

Toute commande produisant une liste de valeurs peut être placée à la suite du **in** à l'aide d'une sous-exécution.

¥ Avec la variable `IFS` contenant `$'\t\n'`, la boucle `for` prendra chaque mot du résultat de cette commande comme liste d'éléments sur laquelle boucler.

¥ Avec la variable `IFS` contenant `$'\t\n'` (c'est-à-dire sans espace), la boucle `for` prendra chaque ligne du résultat de cette commande.

Cela peut être les fichiers d'un répertoire. Dans ce cas, la variable prendra comme valeur chacun des mots des noms des fichiers présents :

```
for fichier in $(ls -d /tmp/*)
do
  echo $fichier
done
```

Cela peut être un fichier. Dans ce cas, la variable prendra comme valeur chaque mot contenu dans le fichier parcouru, du début à la fin :

```
$ cat mon_fichier.txt
première ligne
seconde ligne
troisième ligne
$ for LIGNE in $(cat mon_fichier.txt); do echo $LIGNE; done
première
ligne
seconde
ligne
troisième
ligne
```

Pour lire ligne par ligne un fichier, il faut modifier la valeur de la variable d'environnement `IFS`.

```
$ IFS=$'\t\n'
$ for LIGNE in $(cat mon_fichier.txt); do echo $LIGNE; done
première ligne
seconde ligne
troisième ligne
```

Tester vos connaissances

Toute commande renvoie obligatoirement un code de retour à la fin de son exécution :

\$ Vrai

\$ Faux

Un code de retour `0` indique une erreur d'exécution :

\$ Vrai

\$ Faux

Le code de retour est stocké dans la variable \$?

\$ Vrai

\$ Faux

La commande **test** permet de :

\$ Tester le type d'un fichier

\$ Tester une variable

\$ Comparer des numériques

\$ Comparer le contenu de 2 fichiers

La commande **expr** :

\$ Concatène 2 chaînes de caractères

\$ Effectue des opérations mathématiques

\$ Affiche du texte à l'écran

La syntaxe de la structure conditionnelle ci-dessous vous semble-t-elle correcte ? Expliquez pourquoi.

\$ Vrai

\$ Faux

```
if commande
Ê  commande si $?=0
else
Ê  commande si $?!=0
fi
```

Que signifie la syntaxe suivante : **\${variable:=valeur}**

\$ Affiche une valeur de remplacement si la variable est vide

\$ Affiche une valeur de remplacement si la variable n'est pas vide

\$ Affecte une nouvelle valeur à la variable si elle est vide

La syntaxe de la structure alternative conditionnelle ci-dessous vous semble-t-elle correcte ? Expliquez pourquoi.

\$ Vrai

\$ Faux

```
case $variable in
  valeur1)
    commandes si $variable = valeur1
  valeur2)
    commandes si $variable = valeur2
  *)
    commandes pour toutes les valeurs de $variable != de valeur1 et valeur2
  ;;
esac
```

Parmi les propositions ci-dessous, laquelle n'est pas une structure pour faire une boucle :

\$ while

\$ until

\$ loop

\$ for

La commande `true` renvoie toujours 1:

\$ Vrai

\$ Faux

Chapitre 3. TP Scripting shell

Votre entreprise a besoin d'une solution sécurisée permettant aux personnels de la supervision d'intervenir dans un cadre maîtrisé sur les serveurs.

3.1. Étude du besoin

Votre responsable vous demande de développer un outil destiné aux superviseurs. Ils pourront effectuer quelques actions d'administration ainsi que les premiers diagnostics avant de faire intervenir le personnel d'astreinte.

Le personnel doit pouvoir se connecter aux serveurs via un compte graphique : supervision.

Lorsque l'utilisateur se connecte, un menu est proposé, lui permettant :

¥ De gérer les utilisateurs :

! afficher le nombre d'utilisateurs du serveur et les afficher sous formes de 2 listes :

" les utilisateurs systémes,

" les utilisateurs standards ;

! afficher les groupes du serveur et les afficher sous forme de 2 listes :

" les groupes systémes,

" les groupes standards ;

! créer un groupe : le superviseur devra fournir le GID ;

! créer un utilisateur : le superviseur devra fournir l'UID, le GID, etc. ;

! changer le mot de passe d'un utilisateur ; l'utilisateur sera forcé de changer son mot de passe lors de sa prochaine connexion.

¥ De gérer les services :

! relancer le serveur apache ;

! relancer le serveur postfix.

¥ De tester le réseau :

! Afficher les informations du réseau (Adresse IP, masque, passerelle, serveurs DNS) ;

! Tester le réseau (localhost, ip, passerelle, serveur distant, résolution DNS).

¥ Actions diverses :

! redémarrer le serveur ;

! quitter le script (l'utilisateur est déconnecté).

Les actions du superviseur devront être renseignées dans les journaux système.

3.2. Consignes

- ¥ Les scripts sont stockés dans /opt/supervision/scripts/ ;
- ¥ Effectuer tous les tests que vous jugerez nécessaires ;
- ¥ Découper le code en plusieurs scripts ;
- ¥ Utiliser des fonctions pour organiser le code ;
- ¥ Commenter le code.

3.3. Pistes de travail

- ¥ L'utilisateur supervision aura besoin des droits sudo pour les commandes nécessitant root.
- ¥ Le système attribue le shell /bin/bash à un utilisateur standard, tentez d'attribuer votre script à la place !
- ¥ Utilisez la commande logger pour suivre les actions des superviseurs.
- ¥ Visitez le site : <https://www.shellcheck.net/>

3.4. Proposition de correction



Le code présenté ci-dessous n'est qu'une ébauche effectuée en TP par des stagiaires après 12 heures de cours de script. Il n'est pas parfait mais peut servir de base de correction ou de départ pour l'élaboration d'un travail plus complet.

Création de l'utilisateur

L'utilisateur doit être créé en remplaçant son shell (option -s) par le script que nous allons créer :

```
useradd -s /opt/supervision/scripts/supervision.sh -g users supervision
```

Il faut autoriser l'utilisateur supervision à utiliser sudo mais seulement pour les commandes autorisées. Pour cela, nous allons créer un fichier /etc/sudoers.d/supervision contenant les directives nécessaires :

```
# Liste les commandes autorisées aux superviseurs
Cmd_Alias SUPERVISION = /sbin/reboot, /sbin/ip

# Autorise le superviseur à lancer les commandes précédentes sans saisir de mot de
passe
supervision    ALL=NOPASSWD: SUPERVISION
```

Menu

Cr  er le fichier /opt/supervision/scripts/supervision.sh et lui donner les droits en ex  cution :

```
mkdir -p /opt/supervision/scripts
touch /opt/supervision/scripts/supervision.sh
chown supervision /opt/supervision/scripts/*
chmod u+x /opt/supervision/scripts/*
```

La m  me op  ration sera effectu  e pour chaque script cr    .

```
#!/bin/bash

# Base des scripts

BASE=$(dirname "$0")
readonly BASE

. $BASE/utils.sh

function print_menu {
    while (true)
    do
        clear
        banner
        warning
        echo "Vous pouvez : "
        echo ""
        echo " => 1) Relancer le serveur"
        echo " => 2) Afficher la conf IP"
        echo " => 3) Tester le reseau "
        echo " => 4) Afficher les utilisateurs"
        echo " => 5) Relancer le service apache"
        echo " => 6) Relancer le service postfix"
        echo " => 0) Quitter ce super programme "
        echo ""
        read -p "Que voulez vous faire : " choix
        echo ""
        case $choix in
            "1")
                sudo reboot
                ;;
            "2")
                $BASE/print-net.sh
                ;;
            "3")
                $BASE/check-net.sh
```

```

Ê ;;
Ê "4")
Ê $BASE/affiche-utilisateurs.sh
Ê ;;
Ê "5")
Ê $BASE/gestion-services.sh "httpd"
Ê ;;
Ê "6")
Ê $BASE/gestion-services.sh "postfix"
Ê ;;
Ê "q" | "Q" | "quitter" | "quit")
Ê exit 0
Ê ;;
Ê *)
Ê echo "Cette fonction n'est pas encore developpee"

Ê esac
Ê pause
Ê done
}
banner

echo "Bienvenue sur votre console d'administration"
echo ""
echo "Vous pouvez effectuer quelques diagnostics avant d'appeler le personnel
d'astreinte"
echo ""
warning

pause

print_menu

exit 0

```

Quelques fonctions utilitaires

Le fichier utils.sh contient des fonctions que nous utiliserons dans chaque script :

[illegible]

Le fichier `net-utils.sh` contient les fonctions liřes au rřseau :

```
#!/bin/bash

#
# Fonction utilitaires du reseau
# Version 1
# Depends de utils.sh

function getGateway {
    Ê gateway=$(sudo ip route | grep default | cut -d" " -f3)
    Ê echo $gateway
}

function getDNS {
    Ê DNS=$(grep "nameserver" /etc/resolv.conf | tail -1 | cut -d" " -f2)
    Ê echo $DNS
}

# Test une adresse IP
function checkIP {
    Ê ip=$1
    Ê msg="Test de l'adresse ip : $ip"
    Ê ping -c 1 $ip 1> /dev/null 2>&1
    Ê printOK "$msg" "$?"
}

# test une resolution DNS
function checkDNS {
    Ê res=$(dig +short www.free.fr | wc -l)
    Ê if test "$res" -gt "0"
    Ê then
    Ê     printOK "La resolution DNS fonctionne" "0"
    Ê else
    Ê     printOK "La resolution DNS ne fonctionne pas" "1"
    Ê fi
}

function getPrefix {
    Ê sudo ip add sh | grep " inet " | grep -v "127.0.0.1" | tr -s ' ' | cut -d" " -f 3 |
    cut -d "/" -f2
}

```

La gestion du réseau

Le fichier print-net.sh :

```
#!/bin/bash

#
# Test du reseau
# Version 1
#
# Arguments :
#
ici=$(dirname "$0")
. $ici/utlils.sh
. $ici/net-utlils.sh

echo "L'adresse IP de votre serveur est      : $(hostname -i)"
echo "L'adresse IP de votre gateway est      : $(getGateway)"
echo "L'adresse IP de votre serveur DNS est : $(getDNS)"
echo -n "Votre prefix est : "
getPrefix

echo ""
```

Le fichier check-net.sh :

```
#!/bin/bash

#
# Test du reseau
# Version 1
#
# Arguments :
#
ici=$(dirname "$0")
. $ici/utlils.sh
. $ici/net-utlils.sh

# Gestion du service fourni en argument
checkIP 127.0.0.1
checkIP $(hostname -i)
checkIP $(getGateway)
checkIP $(getDNS)
checkDNS
```

La gestion des services

```
#!/bin/bash
```

```

#
# Gestion des services
# Version 1
#
# Arguments :
# $1 : le nom du service a relancer
#
. ./utils.sh

# Test l'etat du service
# Si le service est demarre, il propose de le relancer
# Sinon le service est demarre
function startService {
    Ê service=$1
    Ê service $service status 1> /dev/null 2>&1
    Ê status=$?
    Ê if test "$status" = "0"
    Ê then
    Ê     # Le service fonctionne deja
    Ê     # Faut-il le relancer ?
    Ê     echo "Le service $service fonctionne..."
    Ê     read -p "Voulez vous le relancer ? O/N " rep
    Ê     if test "$rep" = "0" -o "$rep" = "o"
    Ê     then
    Ê         # L'utilisateur a demande a le relancer
    Ê         logger "SUPP -> Relance d'apache"
    Ê         msg="Relance du serveur $service"
    Ê         service $service restart 1> /dev/null 2>&1
    Ê         printOK "$msg" "$?"
    Ê     else
    Ê         # L'utilisateur ne veut pas le relancer
    Ê         msg="Le service ne sera pas relance"
    Ê         printOK "$msg" "0"
    Ê     fi
    Ê else
    Ê     # Le service ne fonctionne pas
    Ê     # Demarrage
    Ê     logger "SUPP -> Demarrage d'apache"
    Ê     msg="Lancement du serveur $service"
    Ê     service $service start 1> /dev/null 2>&1
    Ê     printOK "$msg" "$?"
    Ê fi
}

# Gestion du service fourni en argument
service=$1
startService $service

```

L'affichage des utilisateurs

Le fichier affiche-utilisateur.sh :


```
#!/bin/bash

# Extrait du fichier /etc/passwd la liste :
# - des utilisateurs du systeme
# - des utilisateurs standards
# Chaque liste est affiche sur une ligne
#
# Version 1.0
# Date : 24/11/2016

# usersys : la liste des utilisateurs systemes
usersys="Voici la liste des utilisateurs systemes :\n"
# userstd : la liste des utilisateurs standards
userstd="Voici la liste des utilisateurs standard :\n"

# Stocker l'IFS dans une variable
OLDIFS=' $IFS'
# Pour que la commande for fonctionne, il faut supprimer l'espace comme caractere de
separation
IFS=$'\n'
# On boucle sur chaque ligne du fichier /etc/passwd
while read -r ligne
do
    Ê # Isoler l'UID
    Ê uid=$(echo $ligne | cut -d: -f3)
    Ê # Isoler le Nom
    Ê nom=$(echo $ligne | cut -d: -f1)
    Ê # Si uid < 500 => Utilisateur systeme
    Ê if test "$uid" -lt "500"
    Ê then
    Ê     # Ajouter le nom a la liste
    Ê     usersys="${usersys}${nom}, "
    Ê else
    Ê     # Ajouter le nom a la liste
    Ê     userstd="${userstd}${nom}, "
    Ê fi
done < /etc/passwd

# Affichage de la liste
echo -e "$usersys"
echo ""
echo -e "$userstd"

IFS=$OLDIFS
```

Glossaire

BASH

Bourne Again SHell

Index

@

\$#, 20
\$\$, 13
\$?, 13, 26
\$@, 20
\$s, 20
\${x}, 20
&&, 36
=, 11
\n, 17
\t, 17
| |, 36

B

basename, 19
bash, 7
break, 39, 41

C

case, 37
continue, 39
cut, 17

D

dirname, 19
do, 38
done, 38

E

elif, 35
else, 35
env, 13
esac, 37
exit, 27, 39
export, 13
expr, 31

F

false, 40
fi, 35
for, 38, 42

H

HOME, 13
HOSTNAME, 13

I

IFS, 17
if, 34
interprŽteur de commande, 7

L

LOGNAME, 13
let, 32
logger, 15

P

PATH, 13
PS3, 41
PWD, 13

R

REPLY, 41
read, 16
readonly, 12

S

select, 38, 41
set, 13
set -u, 13
shell, 7
shift, 21

T

then, 35
tr, 18
true, 40
typeset, 12, 32

U

USER, 13
USERNAME, 13
unset, 12
until, 38, 40

V

variable, [10](#)

W

while, [38](#), [38](#)