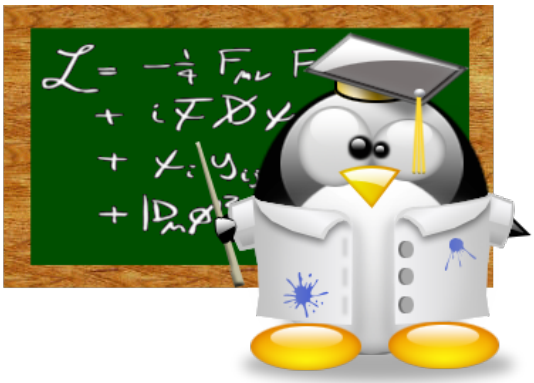

FORMATUX - Support de cours GNU/Linux

Shell

1.0, 24/06/2017

Chapitre 1. 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 la CentOS, qui est le pendant gratuit de la distribution RedHat. La distribution CentOS est particulièrement adaptée pour un usage sur des serveurs d'entreprises.

1.1. Crédits

Ce support de cours a été rédigé par les formateurs :

- Patrick Finet ;
- Antoine Le Morvan ;
- Xavier Sauvignon ;

1.2. Licence

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

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 librements téléchargeables sur framagit :

- https://framagit.org/alemorvan/formatux.fr-support/

Vous y trouverez la dernière version de ce document.

A partir des sources, vous pouvez générer votre support de formation personnalisé. Nous vous recommandons le logiciel AscidocFX téléchargeable ici : http://asciidocfx.com/

1.3. Gestion des versions

Table 1. Historique des versions du document

Version	Date	Observations
1.0	Avril 2017	Version initiale.

Chapitre 2. Les scripts SHELL - Niveau 1

Le shell est l'interpréteur de commandes de Linux. Il ne fait pas partie du noyau, mais forme une couche supplémantaire, d'où son nom de "coquille".

Il a un rôle d'analyse des commandes saisies puis les fait exécuter par le système.

Il existe plusieurs shells, tous partageant des points communs :

- bourne again shell (bash),
- korn shell (ksh),
- c shell (csh),
- etc.

Le bash est présent par défaut sur les distributions Linux, il se caractèrise par ses fonctionnalités pratiques et conviviales.

Le shell est aussi un langage de programmation basique qui grâce à quelques commandes dédiées permet :

- L'utilisation de variables,
- l'exécution conditionnelle de commandes,
- la répétition de commandes.

Les scripts en shell ont l'avantage d'être réalisables rapidement et de manière fiable, sans compilation ni installation de commandes supplémentaires.

Pour écrire un script shell, il suffit de réunir dans un même fichier toutes les commandes nécessaires, une commande par ligne.

En rendant ce fichier exécutable, le shell le lira séquentiellement et exécutera une à une les commandes le comportant.

Lorsque le shell rencontre une erreur, il affiche un message permettant d'identifier le problème mais continue l'exécution du script.

Les erreurs propres aux commandes sont également affichées à l'écran.

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, une seule commande par ligne, les commandes sont aérées…
- portable : le code fonctionne sur tout système Linux, gestion des dépendances, gestion des

droits, etc.

2.1. Premier script

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

vim est un outil adapté a cela.

Le nom du script devra respecter quelques règles :

- pas de majuscule,
- pas de nom de commandes existantes,
- extension en .sh pour indiquer qu'il s'agit d'un script shell.

hello-world.sh

```
#!/bin/bash
#
# Auteur : Antoine Le Morvan
# Date : 18 septembre 2016
# Version 1.0.0 : Affiche le texte "Hello world !"
#

# Affiche un texte à l'écran :
echo "Hello world !"
```

Pour pouvoir exécuter ce script, il est nécessaire de lui donner le droit d'exécution :

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

La première ligne à écrire dans tout script permet d'indiquer le shell à utiliser pour l'exécuter. Si vous désirez utiliser le shell ksh ou le langage interprété python, vous remplaceriez la ligne :

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

par :

```
#!/bin/ksh
```

ou par :

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

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

- un commentaire général en début pour indiquer le but du script, son auteur, sa version, etc.
- des commentaires 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 !
```

2.2. 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. 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 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.

Il n'y a pas de notion de type de variable en script shell. Le contenu d'une variable est toujours un caractère ou une chaîne de caractères.

01-backup.sh

```
#!/bin/bash

#
# Auteur : Antoine Le Morvan
# Date : 18 septembre 2016
# Version 1.0.0 : Sauvegarde dans /root les fichiers passwd, shadow, group et gshadow
#

# Variables globales
FICHIER1=/etc/passwd
FICHIER2=/etc/shadow
FICHIER3=/etc/group
FICHIER4=/etc/gshadow

# Dossier destination
DESTINATION=/root

# Nettoie l'écran :
clear

# Lancer la sauvegarde
echo "La sauvegarde de $FICHIER1, $FICHIER2, $FICHIER3, $FICHIER4 vers $DESTINATION va
commencer :"

cp $FICHIER1 $FICHIER2 $FICHIER3 $FICHIER4 $DESTINATION

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 précaution pour n'être ni trop évasif ni trop compliqué. Une variable peut toutefois être nommée avec des majuscules, comme c'est le cas ici, s'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=valeur
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.

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

```
stagiaire $ touch "$fichier"1
stagiaire $ touch ${fichier}1
```

Supprimer et verrouiller les variables

La commande **unset** permet de supprimer une variable.

Exemple :

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

La commande readonly verrouille une variable.

Exemple :

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

Variables d'environnements

Les variables d'environnements et les variables systèmes sont des variables utilisées par le système

pour son fonctionnement. Par convention elles portent un nom en majuscules.

Elles peuvent être affichées ou modifiées dans un script comme n'importe quelle variable. Elles doivent cependant être modifiées avec précaution.

La commande **env** permet d'afficher toutes les variables d'environnement utilisées.

La commande **set** permet d'afficher toutes les variables système utilisées.

Parmi les dizaines de variables d'environnement, plusieurs ont un intérêt à être utilisées dans un script shell :

Table 2. Variables d'environnement

Variable	Observation
HOSTNAME	Nom d'hôte de la machine
USER, USERNAME et LOGNAME	Nom de l'utilisateur connecté sur la session
PATH	Chemin des commandes
PWD	Répertoire courant, mis à jour à chaque exécution de la commande cd.
HOME	Répertoire de connexion.

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 connaître les variables et leur contenu, il faut les exporter.

La modification d'une variable exportée dans un processus fils ne peut pas remonter au processus père.

Sans option, la commande **export** affiche le nom et les valeurs des variables exportées 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)
```

Exemples :

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

Améliorations du script de sauvegarde

Quelques pistes d'améliorations

```
#!/bin/bash

#
# Auteur : Antoine Le Morvan
# Date : 18 septembre 2016
# 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 a sauvegarder
FICHIER1=/etc/passwd
FICHIER2=/etc/shadow
FICHIER3=/etc/group
FICHIER4=/etc/gshadow

## Dossier destination
DESTINATION=/root

## Variables en readonly
readonly FICHIER1
readonly FICHIER2
readonly FICHIER3
readonly FICHIER4
readonly DESTINATION

# Un nom de dossier avec le quantieme du jour
rep="backup-$(date +%j)"

# Nettoie l'écran :
clear
```

```
# Lancer la sauvegarde
echo "****************************************************************"
echo "     Script de sauvegarde - Sauvegarde sur la machine $HOSTNAME "
echo "****************************************************************"
echo "La sauvegarde sera faîte dans le dossier ${rep}."
echo "Création du répertoire..."
mkdir -p $DESTINATION/$rep
echo "                                                        [ OK ]"
echo "La sauvegarde de ${FICHIER1}, ${FICHIER2}, ${FICHIER3}, ${FICHIER4} vers
${DESTINATION}/$rep va commencer :"

cp $FICHIER1 $FICHIER2 $FICHIER3 $FICHIER4 $DESTINATION/$rep

echo "La sauvegarde est terminée !"

# La sauvegarde est notée dans le journal d'évènements du système :
echo "La sauvegarde est renseignée dans syslog :"
logger "Sauvegarde des fichiers systèmes par ${USER} sur la machine ${HOSTNAME} dans
le dossier ${DESTINATION}/$rep."
echo "                                                        [ OK ]"
```

Exécution de notre script de sauvegarde

```
root # ./02-backup-enhanced.sh
****************************************************************
     Script de sauvegarde - Sauvegarde sur la machine formateur1
****************************************************************
La sauvegarde sera faîte dans le dossier backup-262.
Création du répertoire...
                                                        [ 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 :
                                                        [ OK ]
```

Le lancement de la commande peut être visualisée dans le journal syslog :

Evènement dans syslog

```
root # tail -f /var/log/messages
sept. 18 19:35:35 formateur1 antoine[9712]: Sauvegarde des fichiers systèmes par
antoine sur la machine formateur1 dans le dossier /root/b...
```

2.3. Saisie et manipulations

Selon l'objet du script, il peut être nécessaire d'envoyer des informations en cours d'exécution.

Ces informations, non connues lors de l'écriture du script, peuvent être extraites à 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 chaîne de caractère pour la stocker dans une variable.

Syntaxe de la commande read

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

Exemple de la commande read

```
stagiaire $ read nom prenom
stagiaire $ 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 caractères à saisir
-s	Masque la saisie

Lors de l'utilisation de l'option **-n**, le shell valide automatiquement la saisie au bout du nombre de caractères précisés. L'utilisateur n'a pas à appuyer sur la touche [ENTREE].

```
stagiaire $ read -n5 nom
```

La commande **read** permet d'interrompre l'exécution du script le temps que l'utilisateur saisisse des informations. La saisie de l'utilisateur est découpée en mots affectés à une ou plusieurs variables prédéfinies. 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 prédéfinie.

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

```
[root] # set | grep IFS
IFS=$' \t\n'
```

Par défaut, l'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.

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

La commande cut

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

Syntaxe de la commande cut

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

Syntaxe de la commande cut

```
[root]# 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 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

```
[root]# 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 cut

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 à une seule unité le caractère spécifié

```
[stagiaire]$ tr -s " " < /etc/hosts
```

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

```
#!/bin/bash

#
# Auteur : Antoine Le Morvan
# Date : 18 septembre 2016
# Version 1.0.0 : Extrait le niveau d'exécution du fichier /etc/inittab

# Variables globales

INITTAB=/etc/inittab

niveau=`grep "^id" $INITTAB | cut -d: -f2`

# Affichage du résultat :
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 :

```
[root]# echo $FICHIER=/usr/bin/passwd
[root]# basename $FICHIER
passwd
[root]# 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 s'il s'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 à l'avantage qu'une fois le script exécuté, il n'aura plus besoin d'intervention humaine pour se terminer.

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

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

```
[root]# ./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
$0	contient le nom du script tel qu'il a été saisi.
1 à **$9**	contiennent les valeurs du 1er et du 9ème argument.
${x}	contient la valeur de l'argument **x**, supérieur à 9.
$#	contient le nombre d'arguments passés.
$*	contient en une variable tous les arguments passés

Exemples :

```
[root]# ./script.sh un deux trois
[root]# echo $3 $2 $1
trois deux un
[root]# echo $0 $# $*
./script.sh 3 un deux trois
```

Attention : il existe une autre variable positionnelle, **$@**, qui contient tous les arguments passés. La confusion avec **$*** est aisée.

La différence se fait au niveau du format de stockage des arguments : **$*** : Contient les arguments au format "$1 $2 $3 ..." **$@** : Contient les arguments au format "$1" "$2" "$3" ...

La commande shift

La commande **shift** permet de décaler les variables postionnelles.

Exemples :

```
[root]# ./script.sh un deux trois
[root]# echo $1
un
[root]# shift 2
[root]# echo $1
trois
```

Attention : Lors de l'utilisation de la commande **shift**, les variables **$#** et **$*** sont modifiées en conséquence.

La commande set

La commande **set** découpe une chaîne en variables positionnelles.

Syntaxe de la commande set

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

Exemple :

```
[root]# set un deux trois
[root]# echo $1 $2 $3 $#
un deux trois 3
[root]# variable="un deux trois"
[root]# set $variable
[root]# echo $1 $2 $3 $#
un deux trois 3
```

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

```
#!/bin/bash

#
# Auteur : Antoine Le Morvan
# Date : 18 septembre 2016
# 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
# Version 1.0.2 : Modification pour utiliser les variables positionnelles
#                 Limitation à 5 fichiers

# Variables globales

## Dossier destination
DESTINATION=/root

# Un nom de dossier avec le quantieme du jour
rep="backup-$(date +%j)"

# Nettoie l'écran :
clear

# Lancer la sauvegarde
echo "************************************************************************"
echo "     Script de sauvegarde - Sauvegarde sur la machine $HOSTNAME "
echo "************************************************************************"
echo "La sauvegarde sera faîte dans le dossier ${rep}."
echo "Création du répertoire..."
mkdir -p $DESTINATION/$rep
echo "                                                        [ OK ]"
echo "La sauvegarde de ${1} ${2} ${3} ${4} ${5} vers ${DESTINATION}/$rep va commencer
:"

cp $1 $2 $3 $4 $5 $DESTINATION/$rep

echo "La sauvegarde est terminée !"
```

Chapitre 3. Scripts shell - Instructions de contrôle

3.1. Tests

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

La convention établie veut que si la commande s'est bien exécutée, le code de statut ait pour valeur zéro.

Si la commande a eu un problème lors de son exécution (les raisons peuvent être nombreuses : manque de droits d'accès, absence de fichier, saisie incorrecte, ...), son code de status aura une valeur différente de zéro.

Il faut se référer au manuel de la 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 : **$?**.

```
[stagiaire] $ mkdir repertoire
[stagiaire] $ echo $?
0
[stagiaire] $ mkdir /repertoire
mkdir: impossible de créer le répertoire
[stagiaire] $ echo $?
1
```

L'affichage du contenu de la variable **$?** avec la commande **echo** se fait immédiatement après la commande que l'on souhaite évaluer. Cette variable étant mise à jour après chaque commande.

Il est également possible de créer des codes de retour dans un script. Il suffit pour cela d'ajouter un argument numérique à la commande **exit**.

```
[stagiaire] $ exit 2
[stagiaire] $ echo $?
2
```

Outre la bonne exécution d'une commande, le shell offre la possibilité d'exécuter des tests sur de nombreux motifs :

- Fichiers (existance, type, droits, comparaison) ;
- Chaînes de caractères (longueur, comparaison) ;

- Numériques entiers (valeur, comparaison).

Le résultat du test :

- $?=0 : le test s'est correctement exécuté et est vrai ;
- $?=1 : le test s'est correctement exécuté et est faux ;
- $?=2 : le test ne s'est pas correctement exécuté.

Tester le type d'un fichier

Syntaxe de la commande test pour un fichier

```
test [-d|-e|-f|-L] fichier
```

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 répertoire
-L	Teste si le fichier existe et est de type lien symbolique
-b	Teste si le fichier existe et est de type spécial mode bloc
-c	Teste si le fichier existe et est de type spécial mode caractère
-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 écriture
-x	Teste si le fichier existe et est est exécutable
-g	Teste si le fichier existe et est a un SGID positionné
-u	Teste si le fichier existe et est a un SUID positionné
-s	Teste si le fichier existe et est non vide (taille > 0 octets)

Comparer deux fichiers

La commande **test** peut également comparer des fichiers :

Syntaxe de la commande test pour la comparaison de fichiers

```
test fichier1 [-nt|-ot|-ef] fichier2
```

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

Option	Observation
-nt	Teste si le premier fichier est plus récent qu ele 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 :

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

Table 10. Options de la commande test pour les variables

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

Comparaison de numériques entiers

Syntaxe de la commande test pour les entiers

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

Exemple :

```
[stagiaire] $ test "$var" -eq "1"
[stagiaire] $ echo $?
0
```

Table 11. Options de la commande test pour les entiers

Option	Observation
-eq	Teste si le premier nombre est égal au second
-ne	Teste si le premier nombre est différent au second
-gt	Teste si le premier nombre est supérieur au second
-lt	Teste si le premier nombre est inférieur au second

Les numériques étant traités par le shell comme des caractères (ou chaînes de caractères) classiques, un test sur un caractère peut renvoyer le même résultat qu'il soit traité en tant que numérique ou non.

```
[stagiaire] $ test "1" = "1"
[stagiaire] $ echo $?
0
[stagiaire] $ test "1" -eq "1"
[stagiaire] $ echo $?
0
```

Mais le résultat du test n'aura pas la même signification :

- Dans le premier cas, il signifiera que les deux caractères ont la même valeur dans la table ASCII.
- Dans le second cas, il signifiera que les deux nombres sont égaux.

Combinaison de tests

La combinaison de test 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
```

```
[stagiaire] $ test -d /etc -a -x /etc
[stagiaire] $ 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 groupé 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 :

```
[stagiaire] $ test -e /fichier # test si fichier existe
[stagiaire] $ ! test -e /fichier # test si fichier n'existe pas
```

Les opérations numériques

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

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

Exemple :

```
[stagiaire] $ expr 2 +2
4
```

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

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

La commande typeset

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

Exemple :

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

La commande let

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

Exemple :

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

La commande **let** ne retourne pas un code retour cohérent lorsqu'elle évalue le numérique 0

```
[stagiaire] $ let 0
[stagiaire] $ echo $?
1
```

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

```
[stagiaire] $ let var=5+5
[stagiaire] $ echo $var
10
```

3.2. Structures conditionnelles

Si la variable $? permet de connaître le résultat d'un test ou de l'exécution d'une commande elle ne peut qu'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
    commande si $?=0
else
    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, ...) permet de définir les actions à effectuer en cas de succès ou les messages d'erreur à afficher en cas d'échec.

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

if mkdir rep
then
    cd rep
fi
```

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

```
[...]
else
  if test -e /etc/
[...]

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

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 :

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

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

Cette syntaxe met en oeuvre 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 :

```
[stagiaire]$ nom=""
[stagiaire]$ echo ${nom:-linux}
linux
[stagiaire]$ echo $nom

[stagiaire]$ echo ${nom:=linux}
linux
[stagiaire]$ echo $nom
linux
[stagiaire]$ echo ${nom:+tux}
tux
[stagiaire]$ echo $nom
linux
```

Structure alternative conditionnelle case

Une succession de structures **if** peut vite devenir lourde et complexe. Lorsquelle 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** / **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 $ in
  valeur1)
    commandes si variable = valeur1
    ;;
  valeur2)
    commandes si variable = valeur2
    ;;
  [..]
  *)
    commandes pour toutes les valeurs de variable != de valeur1 et valeur2
    ;;
esac
else
    commande si $?!=0
fi
```

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"
  ;;
```

3.3. 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 repéter 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 vrai ($? = 0), les commandes placées entre **do** et **done** sont exécutées. Le script retourne ensuite au début évaluer de nouveau la commande.

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

Syntaxe de la structure boucle conditionnelle while

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

Exemple :

```
while test -e /etc/passwd
do
  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 dans le script. Cela peut être volontaire, mais aussi être une erreur. Il faut donc faire très attention à la commande qui régit la boucle et trouver un moyen d'en sortir.

Pour sortir d'une boucle while, il faut faire en sorte que la commande évaluée 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 :

```
[stagiaire]$ exit 1
[stagiaire]$ 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.

```
[stagiaire]$ true
[stagiaire]$ echo $?
0
[stagiaire]$ false
[stagiaire]$ 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 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 while

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

Exemple :

```
until test -e /etc/passwd
do
  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 elément 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" va permettre de demander à l'utilisateur de faire un 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
  commandes
done
```

Exemple :

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

ce qui donne à 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 :

```
for fichier in /home /etc/passwd /root/fic.txt
do
  file $fichier
done
```

Toute commande produisant une liste de valeurs peut être placée à la suite du **in** à l'aide d'une sous-exécution. La boucle **for** prendra le résultat de cette commande comme liste d'eléments sur laquelle boucler.

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

```
for fichier in `ls /root`
do
  echo $fichier
done
```

Cela peut être les lignes d'un fichier. Dans ce cas, la variable prendra comme valeur chacune des lignes du fichier parcouru, du début à la fin :

```
for ligne in `more /etc/hosts`
do
  echo $ligne
done
```

Chapitre 4. 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.

4.1. Etude 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 générique : **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 systèmes,
    - les utilisateurs standards ;
  - afficher les groupes du serveur et les afficher sous forme de 2 listes :
    - les groupes systèmes,
    - 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 systèmes.

4.2. 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.

4.3. Pistes de travail

- L'utilisateur supervision aura besoin des droits sudo pour les commandes réservées à 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/

4.4. 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 autorisees aux superviseurs
Cmnd_Alias SUPERVISION = /sbin/reboot, /sbin/ip

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

Menu

Créer le fichier /opt/supervision/scripts/supervision.sh et lui donner les droits en exécution :

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

La même opération sera effectuée pour chaque script créé.

```
#!/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 "  => Q) 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
```

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

    esac
    pause
  done
}
banner

echo "Bienvenue sur votre console d'administration"
echo ""
echo "Vous pouvez effectuer quelques diagnostics avantd'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 :

```
#!/bin/bash
#
# Fonctions utilitaires et variables globales
# Version 1
#
ok="                                          [OK]"
nok="                                         [NOK]"

# Affiche ok ou nok
# Arguments :
# $1 = 0 ou 1
# $2 = message a imprimer
function printOK {
  echo "$1"
  if test "$2" = "0"
  then
    echo "$ok"
  else
    echo "$nok"
  fi
}

function banner {
echo "*            Bienvenue dans l'outil de la           *"
echo "               S U P E R V I S I O N                "
echo "                                    _     _           "
echo " ___ _   _ _ __   ___ _ ____   _(_)___(_) ___  _ __  "
echo "/ __| | | | '_ \ / _ \ '__\ \ / / / __| |/ _ \| '_ \ "
echo "\__ \ |_| | |_) |  __/ |   \ V /| \__ \ | (_) | | | |"
echo "|___/\__,_| .__/ \___|_|    \_/ |_|___/_|\___/|_| |_|"
echo "          |_|                                        "
}



function pause {
  echo "Appuyer sur Entree pour continuer..."
  read
}

function warning {
  echo "ATTENTION !!!"
  echo "Toutes les actions entreprises sont renseignees dans le journal d'evenement !"
  echo ""
}
```

Le fichier net-utils.sh contient les fonctions liées au réseau :

```
#!/bin/bash

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

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

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

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

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

function getPrefix {
  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/utils.sh
. $ici/net-utils.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/utils.sh
. $ici/net-utils.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 {
  service=$1
  service $service status 1> /dev/null 2>&1
  status=$?
  if test "$status" = "0"
  then
    # Le service fonctionne deja
    # Faut-il le relancer ?
    echo "Le service $service fonctionne..."
    read -p "Voulez vous le relancer ? O/N " rep
    if test "$rep" = "O" -o "$rep" = "o"
    then
      # L'utilisateur a demande a le relancer
      logger "SUPP -> Relance d'apache"
      msg="Relance du serveur $service"
      service $service restart 1> /dev/null 2>&1
      printOK "$msg" "$?"
    else
      # L'utilisateur ne veut pas le relancer
      msg="Le service ne sera pas relance"
      printOK "$msg" "0"
    fi
  else
    # Le service ne fonctionne pas
    # Demarrage
    logger "SUPP -> Demarrage d'apache"
    msg="Lancement du serveur $service"
    service $service start 1> /dev/null 2>&1
    printOK "$msg" "$?"
  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
  # Isoler l'UID
  uid=$(echo $ligne | cut -d: -f3)
  # Isoler le Nom
  nom=$(echo $ligne | cut -d: -f1)
  # Si uid < 500 => Utilisateur systeme
  if test "$uid" -lt "500"
  then
    # Ajouter le nom a la liste
    usersys="${usersys}${nom}, "
  else
    # Ajouter le nom a la liste
    userstd="${userstd}${nom}, "
  fi
done < /etc/passwd


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

IFS=$OLDIFS
```

Chapitre 5. Glossaire

BASH
: Bourne Again SHell

BIOS
: Basic Input Output System

CIDR
: Classless Inter-Domain Routing

Daemon
: Disk And Execution MONitor

DHCP
: Dynamic Host Control Protocol

DNS
: Domain Name Service

FQDN
: Fully Qualified Domain Name

LAN
: Local Area Network

NTP
: Network Time Protocol

nsswitch
: Name Service Switch

POSIX
: Portable Operating System Interface

POST
: Power On Self Test

SHELL
: En français "coquille". Á traduire par "interface système".

SMB
: Server Message Block

SMTP

Simple Mail Tranfer Protocol

SSH

Secure SHell

TLS

Transport Layer Security, un protocole de cryptographie pour sécuriser les communications IP.

TTY

teletypewriter, qui se traduit téléscripteur. C'est la console physique.

UEFI

Unified Extensible Firmware Interface

Chapitre 6. Index