Cours Système n°1: le Shell

Éric Gaudefroy

École Hexagone
Cursus Cyberdéfense - M1

1. Généralités

1.1. les différents shells

Le *shell*, ou interpréteur de commandes en français, est le principal outil permettant à l'utilisateur d'interagir avec l'ordinateur en console texte. Dans ce rôle interactif, le *shell* attend la saisie de commandes, puis les exécute. Par ailleurs, le *shell* sert également à exécuter des scripts contenus dans un fichier texte (mode non interactif).
Sous Unix, l'interpréteur traditionnel est le *Bourne shell* (sh). Dans les systèmes compatibles Unix actuels, il a généralement été remplacé par des interpréteurs plus expressifs, tels que *Bourne again shell* (bash), *Korn shell* (ksh) ou *C shell* (csh).
Les systèmes Windows disposent aussi d'un interpréteur de commande. Le *shell* historique présent depuis MS-DOS est *command.com*. Il cohabite depuis Windows NT avec *cmd.exe*, qui est compatible avec les fonctionnalités récentes de Windows. Enfin, depuis Windows 7, la suite logicielle *Windows PowerShell* intègre une interface en ligne de commande, un langage de script nommé *PowerShell* ainsi qu'un kit de développement.

1.2. Lecture des arguments en ligne de commande

Lors de la saisie d'une commande, les caractères blancs (espace, tabulation, retour chariot) sont interprétés comme des séparateurs d'arguments. L'appel d'une commande ou fonction se fait de la façon suivante :

```
[<blanc(s)>]<nom><blanc(s)><arg1><blanc(s)>...<argn><retour_chariot>
```

Par exemple, la commande suivante appelle le programme **mv** avec deux arguments :

```
mv ./file1 ./newdir/copy1
```

Pour insérer un caractère blanc à l'intérieur d'un argument, il faut utiliser le caractère d'échappement « / » (appelé backslash ou anti-slash) ou des guillemets (cf. 4.2. les substitutions sous bash).
De nombreuses commandes acceptent des options. Par exemple, **-R** indique au programme **ls** qu'il faut lister les fichiers présents de manière récursive. On parle d'options courtes lorsqu'elles sont formées d'un unique caractère (comme **-R**), et d'options longues si elles sont de la forme **--recursive**. Ces options peuvent être concaténées ou dissociées. Ainsi, pour la plupart des commandes qui acceptent des options courtes, par exemple **-r**, **-s** ou **-f** <fichiers>, nous pouvons écrire, de façon équivalente :

```
<cmd> -r -s -f <fichier>
<cmd> -rs -f <fichier>
<cmd> -rsf <fichier>
```

1.3. Micro manuel de survie

Voici quelques commandes basiques nécessaires à l'utilisation de la ligne de commande sous Unix. Les tableaux suivants ne présentent qu'une partie des nombreuses commandes disponibles depuis le *shell*. De plus, tous ces éléments seront revus pendant le module.

Arborescence et répertoire courant

Dans un système Unix, « tout est fichier » [1]. À un instant donné, l'ensemble des fichiers accessibles représente une hiérarchie unique de répertoires contenant des fichiers ou d'autres répertoires. La racine de cette arborescence est « / ».
On peut alors désigner les éléments de cette hiérarchie de façon absolue (un chemin absolu commence par //) ou de façon relative. Dans ce second cas, le chemin est décrit depuis le répertoire courant. En effet, comme nous le verrons dans le cours sur les processus, chaque processus évolue dans un répertoire courant (aussi appelé répertoire de travail ou *working directory*). Les commandes permettant de connaître le répertoire de travail ou de le modifier sont les suivantes :

Commande	Effet
pwd	Affiche le répertoire courant.
cd dir	Change le répertoire courant (qui devient *dir*).
cd ..	Change le répertoire courant pour le répertoire parent.

Gestion des répertoires

Commande	Effet
mkdir dir	Crée un répertoire nommé *dir*.
rmdir dir	Supprime le répertoire *dir* si il est vide.
ls	Affiche la liste des fichiers du répertoire courant.

Gestion des fichiers

Commande	Effet
cat file	Affiche le contenu du fichier *file*.
vi file	Ouvre le fichier *file* dans l'éditeur de texte **vi** [2].
file f	Donne des informations sur le fichier *f* en inspectant son contenu.
cp file newfile	Crée une copie du fichier *file* nommée *newfile*.
cp f1 f2 dir	Copie les fichiers *f1* et *f2* dans le répertoire *dir*.
mv file newfile	Déplace le fichier *file* vers *newfile*. Il est ainsi renommé.
mv f1 f2 dir	Déplace les fichiers *f1* et *f2* dans le répertoire *dir*.
rm file	Supprime le fichier *file*.

Nota bene : les commandes **rm**, **mv** et **cp** peuvent mener à la perte de données si les noms transmis en arguments sont erronés. Par défaut, ces commandes ne demandent pas de confirmation lors de la

1. Il existe quelques exceptions, dont les interfaces réseau (voir le cours sur les fichiers).
2. **vi** est un éditeur peu intuitif, mais il est indispensable de savoir l'utiliser car il est présent sur de très nombreux systèmes. D'autres éditeurs peuvent être disponibles, tels que **nano** ou **emacs**.

suppression. Il est donc conseillé de faire attention lors de leur manipulation, ou d'utiliser l'option **-i** (interactive) qui demande confirmation pour toute opération dangereuse.

1.4. Recherche d'informations

man [<n>/-a] <fonction> : page de manuel pour la fonction ou la commande (C ou *shell*) ; on peut aussi imposer le numéro de la section du manuel à consulter (<n>) ou demander à voir toutes les pages portant le nom recherché (-a).
Les pages de manuel sont divisées en différentes sections. Voici les sections standards :

- 1 : programmes et commandes *shell*
- 2 : appels systèmes
- 3 : fonctions de bibliothèques C
- 4 : Fichiers spéciaux
- 5 : Formats de fichiers
- 6 : Jeux
- 7 : Divers
- 8 : Commandes d'administration

Nota bene : Pour accéder aux pages de manuel POSIX, indiquer 1p pour les commandes shell et 3p pour les fonctions C.
info <fonction> : idem, voire plus complet que **man**.
apropos <mot-clé> : liste de pages d'aide relatives à un sujet.
help <commande> : aide relative à une commande interne ou un mot-clé de *bash* (voir aussi **man bash**).
/usr/share/doc/<paquetage>/ : informations spécifiques à un paquetage installé (Linux).

1.5. Recherche de fichiers

which <commande> : emplacement du programme qui serait lancé si la commande était invoquée.
find [<rep-racine>] [<criteres>] : recherche des fichiers selon certains critères. En voici quelques exemples :

- **-name 'bin'** : le nom du fichier doit être bin ;
- **-type d** : la recherche ne porte que sur les répertoires ;
- **-mmin -5** : le fichier doit avoir été modifié pendant les dernières 5 minutes.

Remarque : Par défaut, **find** affiche la liste des résultats trouvés. Il peut aussi être employé pour lancer des commandes à appliquer aux résultats de la recherche : **find <rep-racine> <critères> -exec <cmd> ';'** où l'expression « {} » peut être utilisé dans **<cmd>** pour désigner le fichier trouvé. Une commande utile à combiner avec **find** est **xargs**.
locate <motif> : recherche de fichiers sur le système à partir des indications d'une base de données des fichiers présents. Cette base de données est mise à jour par une tâche périodique, ou manuellement à l'aide de la commande **updatedb**.
Remarque : **locate** est plus rapide que **find** mais la base de données sur laquelle il s'appuie n'est pas forcément à jour. Par ailleurs, certaines versions de **locate** ne prennent pas en compte les permissions, et n'importe quel utilisateur peut donc obtenir par **locate** de l'information sur des fichiers auxquels il n'a pas accès.

1.6. Visualisation de documents

Pour chaque type de fichiers, il existe généralement de nombreux outils de visualisation. Voici quelques exemples :

Type de fichier	Extension	Programmes
PostScript	.ps	gv, ghostview, evince ou okular
Portable Document Format	.pdf	xpdf, evince, okular (ou acroread, propriétaire)
Fichiers texte	.txt par ex.	more, less (ou des éditeurs de texte)
Fichiers bureautique	.doc ou .ods par ex.	loffice

Nota : les fichiers texte peuvent être encodés de différentes manières. L'encodage ASCII (**man 7 ascii**), le plus spartiate, est le dénominateur commun à tous les encodages. Il contient 128 caractères (il utilise 7 bits), parmi lesquels ne figurent pas les accents par exemple. Parmi les encodages utilisés pour représenter la langue française, avec ses accents et ses cédilles, on a généralement le choix entre la norme ISO-8859-1 (aussi appelé *latin 1*)[3] et l'UTF-8. Le traitement de ces différents encodages entre des systèmes différents peut être délicat. L'outil **iconv** permet de traduire des fichiers exprimés dans un encodage vers un autre encodage. Certaines versions de **iconv** peuvent de plus réaliser des traductions intelligentes des caractères n'ayant pas d'équivalent avec l'option //**TRANSLIT** (voir la page de manuel pour plus d'informations).

1.7. Compression (principaux formats)

Type de fichier	Extension	Programmes
gzip	.gz ou .tgz	gzip et gunzip
bzip2	.bz2 ou .tbz2	bzip2 et bunzip2
ZIP	.zip	zip et unzip
RAR	.rar	rar et unrar

Remarque : Doug McIlroy, l'inventeur des pipes Unix (voir section 4.4) a résumée ainsi la philosophie Unix : « *Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams,because that is a universal interface.* » (« Il faut écrire des programmes qui font une chose et qui la font bien. Des programmes qui travaillent ensemble. Des programmes qui travaillent avec des formats texte, car ils représentent une interface universelle. »). Conformément à cette philosophie, les programmes **gzip** et **bzip2** ne savent faire qu'une chose simple : compresser un fichier. Si l'on souhaite archiver et compresser tout un répertoire, il faut les combiner avec **tar**, présenté dans la section suivante.

1.8. Archives TAR (avec ou sans compression)

Les principaux types d'archives TAR sont les suivantes :

3. Une réactualisation a été effectuée en 1999 pour ajouter le symbole €, ainsi que quelques lettres supplémentaires (dont oe), à la place de symboles peu utilisés. Cette réactualisation s'appelle ISO-8859-15 (ou latin 9). Au-delà de la langue française, cet encodage contient les caractères nécessaires à plusieurs langues d'Europe de l'ouest (dont Ä, ü, etc.).

Type de fichier	Extension
Archive non compressée	.tar
Archive compressée avec gzip	.tar.gz ou .tgz
Archive compressée avec bzip2	.tar.bz2 ou .tbz2

Les commandes principales sont :

- extraction : **tar [z/j]xvf <archive>**
- création : **tar [z/j]cvf <archive> <fichiers>**
- examen : **tar [z/j]tvf <archive>**

Le détail des options est donné dans le tableau suivant :

Option	Description
-c -x -t	création extraction test
-z -j	compression gzip compression bzip2
-v	mode verbeux (affiche les noms de fichier)
-f <archive>	spécifie l'archive à créer/extraire/tester

Nota : de nombreuses commandes proposent une option –verbose (ou -v) permettant d'obtenir plus d'informations sur le déroulement des opérations effectuées. On traduira verbose par « bavard » ou « verbeux » en français. Il existe parfois une gradation dans la quantité d'informations affichées : par exemple -vv en affichera plus que -v.

2. Traitement de flux de données

2.1. Entrées/sorties par défaut des programmes

Par défaut, un programme dispose des entrées/sorties suivantes dès son démarrage :

- *stdin* (descripteur de fichier n°0), l'entrée standard ;
- *stdout* (descripteur n°1), la sortie standard ;
- *stderr* (descripteur n°2), la sortie d'erreur (messages d'erreur).

Remarque : Généralement, les sorties standard et d'erreur sortent sur la même console. La distinction entre ces deux sorties permet de séparer les flux par d'éventuelles redirections.
Remarque : Sous Linux, les descripteurs de fichiers d'un processus **pid** sont visibles dans le répertoire **/proc/<pid>/fd**, ou à travers la commande **lsof**. Sous OpenBSD, cette information est disponible avec la commande **fstat**.

2.2. Production de flux

cat [<fichiers>] : recopier sur la sortie standard.
echo [-n] [-e] <message> : afficher sur la sortie standard.
-n : pas d'ajout de retour chariot ;
-e : remplacer les séquences comme \n, \t, \r, \f ou \<ooo> (notation octale) par les caractères spéciaux correspondants.

printf <**format**> <**args**> : afficher des informations sur la sortie standard, de manière semblable à la fonction du langage C.

2.3. Absorption de flux

read [-s] [<**nom1**> **[...** <**nomk**>**]]** : lire une ligne sur l'entrée standard et affecter son contenu à une ou plusieurs variables.

-s : ne pas afficher les caractères entrés au clavier (suppression de l'écho).

read permet ainsi de briser un flux que l'on peut alors traiter ligne par ligne :

```
<cmd> | while read LINE; do
    <traitement>;
done;
```

Remarque : la liste des séparateurs utilisée par **read** est définie par la variable d'environnement **IFS**, qui par défaut contient l'espace, la tabulation et le retour chariot. Ce sont ces séparateurs qui sont utilisés pour découper la ligne et affecter les morceaux aux différentes variables. Les blancs situés en début de ligne sont ignorés. S'il y a plus de morceaux que de variables, la dernière récupère l'ensemble des morceaux restants.

Remarque : le caractère « \ » peut être utilisé au moment de la saisie pour qu'un blanc ne soit pas considéré comme un séparateur. L'option **-r** permet de traiter le caractère « \ » comme un caractère normal.

Remarque : **read -a** <**nom**> affecte les morceaux aux éléments d'une variable de type matrice. Voir section 3.1 pour la manipulation des matrices.

wc [-l] [<**fichier**>**]** : compter le nombre de lignes d'un fichier ou de l'entrée standard.

tee <**fichier**> : journalisation de l'entrée standard dans un fichier et réémission sur la sortie standard.

2.4. Sélection d'une partie du flux

head [-n <**nombre**>**] [**<**fichier**>**]** : affichage des n premières lignes (n=10 par défaut).

tail [-f] [-n <**nombre**>**] [**<**fichier**>**]** : affichage des n dernières lignes (n=10 par défaut).

-f : affichage en continu des nouvelles lignes (utile pour le suivi des journaux).

cut [-d <**delim**>**] [-f** <**champs**>**] [**<**fichier**>**]** : suppression de champs.

-d <**delim**> : choix du caractère délimitant les champs (« \t » par défaut) ;

-f <**champs**> : choix des champs à conserver (-f 2 ou -f 3-4 par ex.).

Voir aussi **grep** (section 5.2).

2.5. Édition de flux

tr <**chars1**> <**chars2**> : remplacement (un pour un) des caractères de la première liste par ceux de la seconde.

tr [-c] -d <**chars**> : suppression des caractères listés.

-c : la suppression porte sur tous les caractères sauf sur ceux listés.

tr -s <**chars**> : réunion des caractères dupliqués parmi ceux listés.

Remarque : **tr** supporte les mêmes notations que **echo -e** pour la représentation de caractères spéciaux, ainsi que les groupes prédéfinis. Par exemple, **tr '[:lower :]' '[:upper :]'** remplace les

minuscules par les majuscules.
Voir aussi **sed** (section 5.3) et **awk** (section 5.4).

2.6. Tri

sort [-n] [-r] [-u] [-t <delim>] [-k <clé-tri-1>[,<clé-tri-n>]] : tri par champs
- **-n** : réalise un tri numérique au lieu d'une comparaison lexicographique ;
- **-r** : inverse le tri ;
- **-u** : supprime les doublons (sur les arguments de tri !) ;
- **-t** : choix du caractère délimitant les champs ;
- **-k** : précise sur quels champs trier (numéro de début et éventuellement numéro de fin).

tac [<fichier>] : affichage des lignes en ordre inverse (Linux).
Remarque : Les fonctions de tri ne peuvent pas effectuer leurs traitements ligne par ligne. Elles ont besoin d'appréhender la totalité du flux pour cela, et peuvent donc avoir un impact sur les performances, s'il y a beaucoup de lignes à traiter.

3. Le shell, un vrai langage de programmation

Certaines constructions offertes par bash sont très puissantes et permettent une grande expressivité. Cependant, si on souhaite écrire des scripts portables, il faut prendre garde car ces constructions ne sont pas disponibles sur tous les systèmes. La commande **checkbashisms** sert à vérifier que de telles constructions ne sont pas utilisées dans un script.

3.1. Variables et fonctions

3.1.1. Manipulation

[declare / export] <nom>=<valeur> : affectation de variables simples
- **export** : passer la variable aux programmes lancés (variable d'environnement).

<nom>=<valeur> <cmd> : lancer une commande dans un environnement modifié (bash).
<nom>[<indice>] : adresser un élément d'une variable de type « matrice ».
export [-p] : voir toutes les variables exportées.
set (sans argument) : voir toutes les variables définies.
unset <nom> : supprimer une variable.
export -n <nom> : ne plus transmettre une variable aux programmes lancés.
$<nom> ou ${<nom>} : valeur de la variable [4]
[function] <nom> () { <instructions> ; } : définition de fonctions (commandes).
unset -f <nom> : suppression de fonction.
export -f <nom> : export de fonction.
export -p -f : affichage de toutes les fonctions exportées.
local <nom>[=<valeur>] : déclarer une variable locale (dans une fonction).
Nota : Par défaut, toutes les variables sont globales.
Nota : Il est possible d'ajouter des attributs aux variables avec la commande **declare** (voir **help declar**e). Par exemple, **declare -r VAR=123** crée une variable en lecture seule.

4. **${<nom>[<indice>]}** permet de récupérer la valeur d'un élément d'une variable de type matrice. On peut aussi récupérer l'ensemble des éléments via **${<nom>[*]}** ou **${<nom>[@]}** (mêmes règles de substitution que pour $* et $@, cf. section 3.1.2).

3.1.2. Variables particulières

Variable	Description
$PATH	chemins de recherche des exécutables, séparés par des « : »
$PWD	répertoire courant
$PS1	invite du *shell*
$$	*pid* du *shell* en cours (cf. cours sur les processus)
$PPID	*pid* du père du *shell* en cours (idem)
$!	*pid* du dernier processus lancé en arrière-plan
$?	code de retour de la dernière commande d'avant-plan (0 indique un succès, toute autre valeur un
$0	nom d'invocation du *shell* (ou script) en cours
$1, $2...	arguments d'invocation du *shell*, du script ou de la fonction en cours
$*	tous les arguments d'invocation du *shell* (ou script) ou de la fonction en cours (hors $0)
$@	idem, à la différence près que « $* » représente « $1 $2 ... » (substitution naturelle) alors que « $
$#	nombre d'arguments d'invocation du *shell* (ou script) ou de la fonction en cours (hors $0)

Les chemins présents dans la variable **PATH** ne doivent désigner que des répertoires de confiance, afin d'éviter qu'un programme légitime soit masqué par un autre programme de même nom. Il est ainsi dangereux de mettre . (le répertoire courant) dans la variable **PATH**.
La commande **shift [<n>]** décale tous les arguments d'un (ou de <n>) rang(s) vers la gauche, sans affecter $0 : $1 est perdu, $2 devient $1, etc...
Remarque : La section 4.2 détaille le comportement des substitutions, et indique en particulier que les remplacements de variables et autre substitutions ont lieu avant l'exécution de la commande.

3.2. Tests

Les tests sont silencieux : le résultat du test n'est pas envoyé sur la sortie standard, mais à travers la valeur de retour, une valeur entre 0 et 255 que chaque commande renvoie à la fin de son exécution. Un code de retour nul correspond à vrai, un code de retour non nul à faux.
test (ou **[ ]**) est une commande permettant de tester certaines conditions sur les fichiers ou les variables d'environnement. L'ensemble des tests disponibles est décrit dans la page de manuel du *shell* et est donné par **help test** sous *bash*. En voici quelques exemples :

- **-f <fichier>** : existence du fichier ;
- **-d <fichier>** : existence du répertoire ;
- **-x <fichier>** : existence + exécutabilité ;
- **<int1> -eq <int2>** : égalité d'entiers ;
- **<int1> -gt <int2>** : stricte supériorité ;
- **<str1>** = **<str2>** : égalité de chaînes de caractères ;
- **-n <string>** : test de chaîne non vide ;
- **-z <string>** : test de chaîne vide ;
- **! <condition>** : négation logique ;
- **<cond1> -a <cond2>** : « et » logique ;
- **<cond1> -o <cond2>** : « ou » logique ;
- /.../ : groupage logique de conditions.

[[ ]] est identique, aux différences suivantes près :

- **<str1>** == **<str2>** : correspondance d'une chaîne à un motif (str2 est traité selon les règles applicables aux noms de fichiers (cf. 4.2.1), à l'exception près qu'un « . » situé en début de chaîne est traité normalement) ;
- **<cond1>** **&&** **<cond2>** : « et fg logique ;
- **<cond1>** || **<cond2>** : « ou » logique ;
- **(...)** : groupage logique de conditions ;
- **true** : toujours vrai (renvoie 0) ;
- **false** : toujours faux (renvoie 1) ;
- **!** **<cmd>** : inverser la valeur booléenne de retour (attention au blanc avant <cmd>).

Remarques :

- il est intéressant de noter que [est une commande interne du *shell* (builtin), alors que [[est un mot-clé du *shell* ;
- [[n'est pas toujours présent parmi les commandes internes du *shell* (ce n'est pas le cas dans *sh* par exemple) ;
- les notions de builtin et de mots-clés sont définis dans la section 4.

3.3. Structures de programmation

Branchement conditionnel :

```
if <ret_cmd>; then
    <cmds>;
[else
    <cmds>;]
fi
```

Boucles :

```
while <ret_cmd>; do
    <cmds>;
done
```

```
for <nom_var> in <liste_valeurs>; do
    <cmds>;
done
```

Nota : Avant les mots-clés **then**, **else**, **do** et **done**, on peut indifféremment mettre un « ; » ou un retour chariot :

```
if true; then echo toto; else echo titi; fi
```

est ainsi équivalent à :

```
if true
then echo toto
else echo titi
fi
```

Branchement conditionnel par étude de cas :

```
case <str> in
    <motif1>) <cmds1> ;;
    ...
    <motifn>) <cmdsn> ;;
esac
```

Les motifs sont traités selon les règles applicables aux noms de fichiers (cf. 4.2.1), à l'exception près qu'un « . » situé en début de chaîne est traité normalement. Il est de plus possible de spécifier plusieurs alternatives, séparées par des caractères « | ».
Afin de modifier le comportement à l'intérieur des boucles **while** et **for**, on peut utiliser les commandes suivantes[5] :

- **break [<n>]** : sort de la structure englobante (<n> fois) ;
- **continue [<n>]** : reprendre au début de l'itération suivante de la boucle (de niveau <n>).

Enfin, les fonctions suivantes permettent de quitter une fonction ou un shell :

- **return [<n>]** : sortir de la fonction (valeur de retour = <n>) ; **exit [<n>]** : sortir du (sous-)shell courant (valeur de retour = <n>).

3.4. Environnement arithmétique

- **((<cmds-calc>))** ou **let <cmds-calc>** : exécution en mode calcul ;
- **$((<expr>))** ou **$[<expr>]** : substitution arithmétique.

Exemple :

```
$ i=0
$ while [ $i -lt 3 ]; do
> let j=2*i
> echo $((i++)) $j
> done
0 0
1 2
2 4
```

Remarque : Il existe d'autres outils courants pour manipuler les expressions arithmétiques, comme **expr** ou des langages de script génériques comme *perl*.

3.5. Scripting

3.5.1. Rédaction

- **#!** **<cmd>** : choix de l'interpréteur (première ligne du script) ;
- **#<texte>** : commentaire (ligne complète ou fin de ligne) ;
- / ↩ : continuer sur la ligne suivante (sans arrêter la lecture de la commande).

5. Ni les **if** ni les **case** ne sont comptabilisés dans les <n> dont il est question.

3.5.2. Exécution

- **./<script>** : lancement du script **<script>** du répertoire courant[6] ; l'exécution se passe dans un autre *shell* que celui depuis lequel on lance le script ;
- **. <script>** ou **source <script>** : exécution des commandes du script dans le *shell* courant (même si le script n'est pas exécutable), l'éventuelle ligne spécifiant l'interpréteur est ignorée.

source est souvent utilisé pour importer des fonctions ou des variables de configuration (possibilité d'utiliser ces variables/fonctions dans tous les scripts qui les « sourcent »). Certains scripts, comme **~\bashrc** (ou **~\.profile** à la connexion) sont automatiquement « sourcés » au démarrage d'un *bash* en mode interactif.

- **exec <prog>** : finir le *shell* courant en lançant un programme ;
- **exec <redirs>** : appliquer les redirections (voir section 4.3) au *shell* courant et continuer ;
- **exec 2>/dev/null** : élimine la sortie d'erreur.

4. Détails sur le fonctionnement du *shell*

Les commandes exécutables en *bash* sont de différentes sortes disposant de priorités différentes : une commande de priorité haute peut masquer une commande de basse priorité de même nom. De manière décroissante :

- les alias définis (voir ci-après) ;
- les mots-clés du *shell* (**for**, **if**, etc.) ;
- les fonctions définies ;
- les commandes internes du *bash* (builtin) ;
- les programmes exécutables accessibles.

Les programmes peuvent être désignés par un chemin absolu (commençant par « / »), relatif (contenant au moins un « / » mais pas comme premier caractère) ou implicite (ne contenant pas de « / » ; dans ce cas, le programme est recherché dans les répertoires décrits par la variable **$PATH**).
Remarque : La commande interne **type** permet de déterminer quelle est le type d'une commande dans le contexte courant. Ainsi **type type** renvoie normalement « *type is a shell builtin* ».

4.1. Alias

- **alias [-p]** : liste des alias courants.
- **alias <nom>=<commande>** : création d'un alias <nom> vers la commande <commande>.
- **unalias <nom>** : suppression d'un alias.

Voici un exemple simple d'utilisation :

```
$ ls
fichier
$ alias lla='ls -la'
$ lla
total 12
```

6. Celui-ci doit être exécutable, ce qui s'obtient par **chmod +x <script>**.

```
drwxr-xr-x 2 user user 4096 sep 9 22:40 .
drwxrwxrwt 12 root root 4096 sep 9 22:39 ..
-rw-r--r-- 1 user user 8 sep 9 22:40 fichier
$ alias -p
alias lla='ls -la'
```

Protéger le nom d'une commande par des guillemets (simples ou doubles) évite qu'un éventuel alias soit pris en compte mais n'interdit pas qu'une fonction cache la commande demandée. **builtin <cmd>** permet d'exécuter la commande interne du *bash* **<cmd>** en ignorant un éventuel masquage par une fonction (ou un alias). Ceci ne garantit toutefois pas l'absence de masquage de *builtin* lui-même...

4.2. Substitutions

Outre les remplacements de variables, le *shell* effectue un certain nombre de transformations avant d'exécuter la ligne de commande qu'il reçoit.

4.2.1. Noms de fichiers

Le *shell* remplace les arguments contenant les motifs suivants par la liste des fichiers présents correspondant aux critères de recherche (les noms de fichiers sont alors pris en tant qu'arguments séparés). Si aucun fichier ne correspond aux critères, la substitution n'est pas effectuée.

- ***** : nombre quelconque de caractères quelconques ;
- **?** : un caractère quelconque ;
- **[a-z], [0-9], [abc], [A-Z012]** : un caractère parmi une sélection ;
- **[[:upper :]], [[:digit :]]** : un caractère parmi une sélection prédéfinie (*bash*).

Supposons qu'un répertoire contienne les fichiers *img1.jpg, Img2.jpg, img3.png et .config*. Voici quelques exemples de substitutions sur les noms de fichiers :

- * donne img1.jpg Img2.jpg img3.png
- img ?.jpg donne img1.jpg

[:alpha :]mg[2-3]* donne Img2.jpg img3.png

- .* donne . .. .config

Remarque : Les éléments **.** et **..** qui apparaissent dans le dernier cas traité sont présents dans tous les répertoires. Il s'agit respectivement du répertoire courant, et du répertoire parent (voir cours sur les fichiers).

4.2.2. Génériques

- **{<1>,<2>[,...]}** : choix parmi plusieurs alternatives (ex : ***.{jpg,png} ; fic{,.txt,.html}**). Cette substitution a toujours lieu (*bash*).
- ~ : répertoire *home* de l'utilisateur courant (seulement en tête de chemin) ;
- ~**<nom>** : répertoire *home* d'un utilisateur donné (seulement en tête de chemin et si cet utilisateur existe).

4.2.3. Exécution de commandes

- **`<cmd>`** : (caractère « backquote ») remplacé par la sortie standard de la commande ;
- **$(<cmd>)** : idem (ex : **KERNEL=$(uname)**).

Les commandes ainsi exécutées sont lancées dans un « *sous-shell* ». Un *sous-shell* hérite de l'environnement de son père (le *shell* courant) au moment de sa création, y compris des variables et fonctions non exportées. En revanche, les modifications d'environnement ultérieures (variables, répertoire de travail, etc.) faites par l'un n'affectent pas l'environnement de l'autre.

4.2.4. Contrôle des substitutions

- **""** : regroupement[7] et lecture sans interprétation (à l'exclusion des caractères « " », « $ », « / », « ` »). En particulier, les blancs, les symboles de combinaison de commandes ou de redirection (voir sections 4.4 et 4.3) ne sont plus interprétés comme tels.
- **"** : regroupement et lecture sans interprétation (sauf « ' »).
- **¨** : idem, mais le caractère \est interprété et utilisable pour représenter des caractères spéciaux, selon les mêmes notations que **echo -e**, mais aussi en notation hexadécimale *x*<*hh*>.
- **\<carac>** : méta-caractère à lire comme un caractère normal (ex : \$ ou \\).

Par exemple, les commandes suivantes ont le même effet :

```
$ echo "'"'"'"
'"
$ echo \textbackslash ' \textbackslash "
'"
```

eval <expression> : exécution des substitutions dans l'expression <*expression*>, puis réinterprétation de la commande qui en résulte. **eval** permet de faire deux passes de substitutions, généralement en vue d'exploiter des valeurs de variables lors de la réinterprétation (nom de variable ou redirection définies par une variable, etc.). Inversement, ceci présente un risque majeur d'injection de commandes arbitraires lorsque **eval** est utilisé dans un script destiné à utiliser des données fournies par l'utilisateur. En effet, des caractères tels que « ; » peuvent apparaître au cours de la première passe et être pris en compte lors de la deuxième. On a ainsi :

```
$ x="toto ; echo titi"
$ echo "$x"
toto; echo titi
$ eval echo "$x"
toto
titi
```

La commande **echo titi** a été affichée la première fois, et exécutée la seconde. Les versions récentes de *bash* supportent la notation $!<nom> pour désigner la valeur d'une variable dont le nom est contenu dans une autre variable. Il s'agit d'une variante plus sûre, mais moins portable, à l'exécution de **eval** sur l'expression **\$$<nom>**.

7. Le regroupement consiste à considérer comme un seul argument un ensemble de caractères, même s'il y a des caractères blancs.

4.2.5. Impacts imprévus des substitutions

- comment supprimer un fichier nommé « -f » ou « ~ » ?
- si un répertoire comprenant des fichiers *-R* et *fichier*, ainsi qu'un répertoire intitulé *rep*, que renvoie **ls *** exécutée depuis ce répertoire ? Pourquoi ?

Pour répondre à la première question, il existe plusieurs astuces, par exemple préfixer le nom du fichier par le chemin complet du répertoire courant, ou tout simplement par « ./ ». Cependant, le problème soulevé par ces questions dans le cas général nécessite parfois de faire appel à l'argument `--`, supporté par les commandes standards, et qui a la signification suivante : tous les arguments suivant `--` ne devront pas être considérés comme des options.
Remarque : La gestion des options est souvent gérée par la fonction **getopt**. Sous Linux, l'implémentation par défaut de cette fonction s'autorise à réordonner les options passées sur la ligne de commande. Cette flexibilité peut être agréable (**ls /tmp -l** est transformé en **ls -l /tmp**), mais elle démultiplie l'impact imprévu des substitutions discuté ci-dessus. Pour supprimer ce comportement laxiste de **getopt**, il est possible de définir la variable d'environnement **POSIXLY_CORRECT** (voir la page de manuel de **getopt**).

4.3. Redirections

Avant toute chose, rappelons que les numéros des descripteurs de fichiers sont les suivants : 0 pour *stdin*, 1 pour *stdout* et 2 pour *stderr*.

- <**cmd**> > <**fichier**> : redirection de *stdout* (mode écrasement) ;
- <**cmd**> > /**dev**/**null** : suppression de *stdout* ;
- <**cmd**> **2**> <**fichier**> : redirection de *stderr* (mode écrasement) ;
- <**cmd**> **2**>**&1** : redirection de *stderr* vers *stdout* ;
- <**cmd**> » <**fichier**> : redirection de *stdout* (mode ajout, ou *append*) ;
- <**cmd**> < <**fichier**> : redirection de l'entrée *stdin* ;
- <**cmd**> « <**clé_fin**> : remplacement de l'entrée standard par le texte entré jusqu'à <clé_fin> (que l'on choisit souvent égal à *EOF*[8]) ;
- <(<**cmd**>) : remplace un argument de type « nom de fichier » (pour un accès en lecture) par une commande générant son contenu (sur *stdout*), exécutée dans un *sous-shell* (*bash*) ;
- >(<**cmd**>) : remplace un argument de type « nom de fichier » (pour un accès en écriture) par une commande exploitant son contenu (sur *stdin*), exécutée dans un *sous-shell* (*bash*).

Quelques exemples d'utilisation :

```
cat > script.sh << EOF
#!/bin/sh
echo Test
EOF

dd if=/dev/zero of=/dev/sdc4 2>&1 >/dev/null
```

Le premier exemple montre comment utiliser « pour écrire un script depuis le *shell*. Le second exécute une commande, et remplace la sortie standard par la sortie d'erreur (attention à l'ordre).

8. End Of File.

4.4. Combinaison de commandes

- **<cmd1> ; <cmd2>** : chaînage séquentiel ;
- **<cmd1> && <cmd2>** : <cmd2> lancée si <cmd1> a réussi ;
- **<cmd1> || <cmd2>** : <cmd2> lancée si <cmd1> a échoué ;
- **<cmd1> & <cmd2>** : chaînage parallèle (<cmd1> mise en tâche de fond, dans un *sous-shell*). la valeur de retour de l'ensemble est celle de <cmd2>.
- **<cmd1> | <cmd2>** : (*pipe*) redirection de la sortie standard de <cmd1> sur l'entrée standard de <cmd2> (les deux étant lancées dans des *sous-shell* séparés). Si <cmd1> ou <cmd2> contiennent d'autres redirections, celles liées au *pipe* sont réalisées en premier. La valeur de retour du pipe est celle de <cmd2>.
- **<cmds>[;]** : groupage fonctionnel (par exemple pour délimiter une redirection) ;
- (**<cmds>[;]**) : exécution dans un *sous-shell*.

Remarque : **<cmd1> && <cmd2>** est équivalent, au code de retour près, à **if <cmd1> ; then <cmd2> ; fi**. En effet, les codes de retour diffèrent si les deux commandes renvoient un code de retour faux. En revanche, **<cmd1> || <cmd2>** est exactement équivalent à **if ! <cmd1> ; then <cmd2> ; fi** ou encore à **if <cmd1> ; then true ; else <cmd2> ; fi**. Voir 3.3 pour la définition des structures de contrôle.
Remarque : Avec le pipe (« | »), les deux commandes sont exécutées dans des *sous-shell*. *bash* permet d'utiliser d'autres formes de redirection similaires pour exécuter <cmd1> ou <cmd2> dans le *shell* courant, par exemple en vue d'y récupérer la valeur de certaines variables. Ces constructions sont **<cmd1> > >(<cmd2>)** et **<cmd2> < <(<cmd1>)**.

4.5. Avant-plan / Arrière-plan

- **Ctrl-Z** : mise en sommeil de la tâche d'avant-plan courante ;
- **<cmd> &** : lancement en arrière-plan (dans un *sous-shell*) ;
- **fg [%<n>]** : reprise ou mise en avant-plan (<n> = numéro de tâche *bash*) ;
- **bg [%<n>]** : reprise en arrière-plan (<n>=numéro de tâche bash) ;
- **jobs** : liste des tâches en cours pour le *shell*.

4.6. Transformations de variables (*bash*)

Lors du remplacement d'une variable par sa valeur, **base** permet d'effectuer des transformations sur cette valeur avant de l'exploiter dans la ligne de commande. Les principales transformations possibles sont :

- **${<nom> :<début>[:<taille>]}** : extraction de la sous-chaîne de taille <taille> (ou allant jusqu'au bout de la chaîne) à partir du caractère de rang <début> [9].
- **${#<nom>}** : remplacement de la chaîne par sa taille ;
- **${#<nom>[*]}** ou **${#<nom>[@]}** : remplacement de la matrice par son nombre d'éléments ;
- **${<nom>#[#]<motif>}** : suppression du début de la chaîne s'il correspond au motif, en cherchant la correspondance la plus courte (#) ou la plus longue (##) ;

9. Le premier caractère a pour indice 0.

- **${<nom>/[/]<motif>/<rempl>}** : remplacement d'une (/) ou de toutes (//) les occurrences du motif dans la chaîne par <rempl> [10].

Ainsi, si la variable **FILENAME** contient un chemin absolu (par exemple */usr/bin/gcc*), $FILENAME##*/ correspond au nom du fichier (gcc) et $FILENAME%/* renvoie le nom du répertoire (*/usr/bin*).

5. Les expressions rationnelles (*regex*)

5.1. Généralités

5.1.1. Composition

Les expressions rationnelles (*regular expressions* en anglais, d'où le raccourci *regex*, et l'anglicisme « expressions régulières ») se construisent par juxtaposition de :

- **caractères explicites** : a,b,c, etc... ;
- **caractères dans une sélection** : [abv1], [0-9a-fA-F] ;
- **caractères appartenant à des groupes prédéfinis** : [:alnum :], [:upper :], [:print :], etc... ;
- **caractères absents d'une sélection** : [\^abv1] ;
- **caractère quelconque** : . ;
- facteurs de multiplication du dernier « caractère » (ou chaîne) :
  ? : 0 ou 1 ;
  + : 1 ou plus ;
  * : 0 ou plus ;
  n[,[p]] : n, n ou plus, de n à p.
- **indicateurs de début « ^ » et de fin « $ » de ligne** ;
- **sous-chaînes** : ( ) ;
- **« ou » logique entre chaînes** : |

En cas de possibilités d'interprétation multiples (par exemple à cause d'un multiplicateur *), les correspondances sont en principe prises les plus longues possible. Par exemple, l'expression rationnelle -.*- appliquée à la chaîne de caractère aa - bb - cc - dd - ee désignera a priori - bb - cc - dd -, puisque c'est la plus longue chaîne correspondant au motif.

5.1.2. Modes d'expression

Il existe deux modes de formulation pour les expressions rationnelles :

- **le mode basique** : les métacaractères sont \ ?, \+, \, \, \|, \(et \) ;
- **le mode étendu** : les métacaractères sont ?, +, , , |, (et)

Dans les deux cas, * est toujours un métacaractère (et * représente le caractère « * »). Par exemple, ^[[:upper :]] \+1 en mode basique concerne les lignes commençant (^) par une ou plusieurs (\+) majuscules ([[:upper :]]) suivies d'un 1. L'expression désigne alors le début de la ligne jusqu'au '1' inclus.

10. Un caractère # en début de motif impose que le motif se trouve au début de la chaîne, et un caractère % qu'il se trouve à la fin.

5.2. grep

Grep est l'abréviation de « *global regular expression print* ». Elle permet de rechercher des motifs et des chaînes de caractères dans des fichiers, mais aussi de filtrer les informations recherchées même dans les fichiers journaux volumineux. Son utilisation est la suivante :

```
[e]grep [-r] [-A<n>] [-B<n>] [-v] [-i] [-l] <regex> [<fichiers>]
```

Avec pour options principales :

- **egrep** : expressions étendues (grep : expressions basiques) ;
- **-r** : récursif (répertoires) ;
- **-A<n>** : conserver n lignes après celles qui correspondent ;
- **-B<n>** : conserver n lignes avant celles qui correspondent ;
- **-C<n>** : conserver n lignes avant et après celles qui correspondent ;
- **-v** : inverser la sélection (garder les lignes qui ne correspondent pas au motif) ;
- **-i** : ignorer la casse (majuscules/minuscules) ;
- **-l** : n'afficher que le nom des fichiers où le motif a été trouvé, pas les lignes.

Exemples d'utilisation :

```
$ ( echo "Hello everybody," ; echo "This is an example textfile." ; echo "En
$ grep "^Hello" textfile
Hello everybody,

$ egrep "e+" textfile
Hello everybody,
This is an example textfile.

$ grep "y$" textfile
Enjoy
```

5.3. sed

Sed est l'abréviation de « *stream editor* » dans sa forme abrégée). C'est un éditeur de texte non interactif : aucune modification n'est apportée directement au fichier que vous êtes en train de traiter. Vous créez d'abord un fichier temporaire, puis son contenu est transféré dans le fichier d'origine. La commande **sed** fonctionne ligne par ligne : chaque ligne d'un fichier est lue, traitée et restituée individuellement. La principale fonction de la commande **sed** consiste à rechercher certaines chaînes de caractères pour les remplacer par d'autres caractères. Son utilisation est la suivante :

```
sed [-r] <commandes> [<fichiers>]
```

Avec pour options principales :

- **-r** : regex en mode étendu (versions récentes seulement ; défaut=basique) ;
- **-f <script>** : appliquer la suite de commandes sed incluses dans un fichier script ;
- **<commandes>** : appliquer les commandes sed.

Si l'on souhaite passer les commandes directement sur la ligne de commande, il faut retenir que :

- <commandes> doit être vu comme un seul argument par le *shell* ;

- il faut toujours séparer les commandes **sed** par des retours à la ligne.

Il est aussi possible d'écrire des scripts **sed** exécutables, en utilisant un shebang du type **#!/bin/-sed -f** dans la première ligne, et en rendant le fichier exécutable. Le fonctionnement des scripts **sed** est le suivant :

- suite de couples sélecteur + commande, évalués dans l'ordre ;
- les lignes en entrée (*stdin* ou <fichiers>) sont traitées une par une ;
- pour chaque couple, la ligne est comparée au sélecteur et la commande lui est éventuellement appliquée (tous les couples sont parcourus) ;
- l'action par défaut est de conserver la ligne ;
- principaux sélecteurs :

  (vide) : toutes les lignes

  /<regex>/ : expression rationnelle

  <n> : numéro de ligne

  <sélecteur1>,<sélecteur2> : activation sur <sélecteur1>, désactivation sur <sélecteur2>

  <sélecteur>! : toutes les lignes sauf celles correspondant au sélecteur
- commandes principales :

  d : suppression de la ligne en cours de traitement

  s/motif/rempl/ : substitution de la première occurence

  s/motif/rempl/<n> : substitution de <n> occurences

  s/motif/rempl/g : substitution de toutes les occurences

Quelques compléments sur la commande s :

- le délimiteur « / » séparant les différents motifs peut être remplacé par n'importe quel caractère, mais dans tous les cas, on ne pourra utiliser le délimiteur choisi dans les motifs de recherche et de remplacement sans l'échapper avec un « \ » ;
- le motif de remplacement peut utiliser :

  & : partie de la ligne ayant correspondu au motif

  \1 à \9 : sous-chaînes délimitées par \(\) dans le motif
- les portions de la ligne en dehors du motif de recherche sont reproduites sans modification.

Quelques exemples :

- **sed 's/\///' ou sed 's+/++'** : suppression du premier / d'une ligne ;
- **grep '<regex>'** est équivalent à sed '/<regex>/ !d' si <regex> ne contient pas de « / » ;
- **sed s/"^[[:blank :]]*"//** : suppression des blancs en début de ligne ;
- **sed 's§\([0-9]\1,2\\)/\([0-9]\1,2\\)/ \([0-9]\4\\)§\3/\2/\1§g'** : transformation des dates au format français (JJ/MM/AAAA) au format anglais (AAAA/MM/JJ).

sed est un outil très pratique, mais qui ne dispose pas de variables ni de fonctions. On se tournera donc vers **awk** (section suivante) pour des transformations nécessitant plus d'expressivité.

awk

Awk est une commande puissante et un langage de script à usage général conçu pour le traitement de texte avancé. Il est principalement utilisé comme outil de rapport et d'analyse. Il prend les données d'entrée, le transforme et envoie le résultat à la sortie standard. Son utilisation est la suivante :

```
awk <commandes> [<fichiers>]
```

Avec pour options principales :

- **-f <script>** : appliquer la suite de commandes **awk** incluses dans un fichier script ;
- **<commandes>** : appliquer les commandes awk.

Comme pour **sed**, <commandes> doit être vu comme un seul argument par le *shell*. En revanche, **awk** interprète les expressions rationnelles en mode étendu. Il est aussi possible d'écrire des scripts **awk** exécutables, en utilisant un shebang du type **#!/usr/bin/awk -f** dans la première ligne, et en rendant le fichier exécutable. Le fonctionnement des scripts **awk** est le suivant :

- suite de couples sélecteur + commande évalués dans l'ordre ;
- les lignes (enregistrements) en entrée sont traitées une par une ;
- les enregistements sont parsés en champs ;
- les champs sont accessibles par $1, ...(0 =$ tout l'enregistrement) ;
- pour chaque couple, la ligne est comparée au sélecteur, et la commande lui est éventuellement appliquée (tous les couples sont parcourus) ;
- l'action par défaut est de supprimer la ligne ;
- voici les principaux sélecteurs :

  (vide) : tous les enregistrements

  BEGIN : avant le traitement de la première ligne (initialisation)

  END : après le traitement de la dernière ligne (terminaison)

  /<regex>/ : expression rationnelle

  <sélecteur1>,<sélecteur2> : activation sur <sélecteur1>, désactivation sur <sélecteur2>

  opérations logiques sur des sélecteurs (!, && et ||)
- les commandes principales :

  print : imprimer la ligne en cours

  printf : similaire à la fonction C

  print/printf | "<commande>" : passage à une commande externe (sed par exemple),

  next : fin du traitement de la ligne en cours (passage à la ligne suivante, sans examiner les couples restants).

Exemples :

- **egrep '<regex>'** est équivalent à **awk '/<regex>/print'** si <regex> ne contient pas de caractère « / » ;
- **awk '/2/ print $0'** affiche toutes les lignes où il y a un 2 ;
- **awk '$1/2/ print $0'** affiche toutes les lignes contenant un 2 dans le premier champ.

Remarque : Il existe dans **awk** des variables utilisables dans les sélecteurs. Celles-ci sont persistantes entre le traitement des différentes lignes. De plus, on peut définir des fonctions. Par exemple, **awk 's=s+$1 END print s'** écrit la somme de tous les nombres de la première colonne d'un fichier.
Remarque : Il est possible de modifier le comportement de **awk** en redéfinissant :

- le séparateur d'enregistrements avec la variable RS, qui vaut "\n" par défaut ;
- le séparateur de champs, la variable FS, qui vaut " " par défaut. Celui-ci peut également être défini sur la ligne de commande avec l'option -F.

Ainsi, la commande **awk 'BEGIN {FS = ":"} {print $1 " -> " $6}' /etc/passwd** affichent le login et le répertoire home de tous les comptes de la machine.

Ce document est inspiré du « Cours Système n°1 : le Shell » délivré par monsieur Olivier Levillain dans les cours dispensés à l'Agence nationale de la sécurité des systèmes d'information (Septembre 2013).