

IFOSUP

Institut de Formation Supérieure
Ville de Wavre

Systemes d'exploitation

Cédric Vanconingsloo

Table des matières

1. Introduction	1
2. Powershell	2
2.1. Les raccourcis clavier	2
2.2. L'invite de commande interactive	3
2.3. Commandes de bases de Powershell	4
2.3.1. Utilisation des paramètres	4
2.3.1.1. Les paramètres nommés	4
2.3.1.2. Les paramètres « interrupteurs »	5
2.3.2. Les commandes <i>Location</i>	5
2.3.3. Les commandes <i>Item</i>	5
2.3.4. Les commandes <i>Host</i>	7
2.3.5. Les commandes <i>Computer</i>	8
2.3.5.1. Les commandes liées à la restauration du système	8
2.3.6. Autres commandes utiles	9
2.3.7. Les commandes <i>Service</i>	9
2.3.8. Les commandes <i>User</i>	10
2.4. Les alias	10
2.4.1. Créer un alias	10
2.5. Le Pipeline	11
3. Conception de scripts	12
3.1. Les variables	12
3.1.1. Création de variables	12
3.1.2. Vérifier l'existence d'une variable	13
3.1.3. Les cmdlets de gestion des variables	13
3.1.4. Variables en lecture seule et constante	14
3.1.5. Les variables automatiques	15
3.1.6. Typage des variables	15
3.2. Les conditions	15
3.2.1. Les opérateurs de comparaison	15
3.2.2. Les opérateurs logiques	16
3.2.3. If-ElseIf-Else	16
3.2.4. Switch	17
3.3. Les boucles	18
3.3.1. ForEach et ForEach-Object	18
3.3.2. Les boucles indéfinies	19
3.3.3. La boucle finie	19
4. Introduction à Linux avec ArchLinux	20
4.1. Pourquoi Linux ?	20
4.2. Pourquoi différentes <i>distributions</i> ?	20
4.3. Pourquoi des interfaces graphiques différentes?	20
4.4. Pourquoi ArchLinux?	21
4.4.0.0.1. Que faut-il télécharger?	21
4.5. Tester ArchLinux sans risques	22
4.5.1. La machine virtuelle	22
4.5.1.1. Installer une machine virtuelle	22
4.5.1.2. Installer ArchLinux dans une machine virtuelle	23

4.5.2. Live CD / Live USB	24
4.5.2.1. Créer un live USB	24
4.5.2.2. Créer un live CD	25
4.5.2.3. Utiliser un live CD/USB	25
4.6. L'installation en côte à côte	26
4.7. L'installation en remplacement de Windows.	26
5. Utiliser la console	27
5.1. Les types de shell	27
5.2. Les commandes indispensables	28
5.2.1. Syntaxe d'une commande Linux	28
5.2.2. Les commandes de base	29
5.2.3. Les alias	30
5.2.4. Commandes internes et externes	30
5.3. Les flux de redirection et le piping	30
5.3.1. Redirection des erreurs	31
5.3.2. Piping	31
6. Organisation du système de fichiers	32
6.1. Les fichiers sous Linux	32
6.2. Les systèmes de fichiers	33
6.3. Les chemins	34
6.3.1. Chemin absolu	34
6.3.2. Chemin relatif	35
6.3.3. Chemin personnel	35
6.4. Les points de montage	35
6.5. Les commandes relatives aux fichiers	35
7. Les droits d'accès aux fichiers	38
7.1. Concepts de compte utilisateur et de groupe	38
7.2. Les droits sous Linux	38
7.3. Gestion des droits	38
8. Gestion des processus	41
9. Outils divers	42
9.1. find et locate	42
9.1.1. Chemin de recherche de <code>find</code>	42
9.1.2. Expressions de sélection de <code>find</code>	42
9.1.3. Actions de <code>find</code>	42
9.2. <code>grep</code> , <code>cut</code> , <code>sort</code>	43
10. Notions de shell scripting avec ZSH	44
10.1. Écrire notre premier script	44
10.2. Les variables	45
10.3. La commande <code>read</code>	45
10.4. Les commentaires et les « quotes »	46
10.5. Passage de paramètres	47
10.6. La commande <code>shift</code>	47
10.7. Les tableaux	48
10.8. Les structures de contrôles	48
10.8.1. Les structures <code>&&</code> et <code> </code>	48
10.8.2. La commande <code>test</code>	48

10.8.2.1. Prédicats	48
10.8.3. La commande <code>expr</code>	49
10.8.4. La commande <code>exit</code>	49
10.9. La structure <code>if-elif-else-fi</code>	49
10.10. La structure <code>case-esac</code>	50
10.11. Les structure itératives	51
10.11.1. La boucle <code>while-do-done</code>	51
10.11.2. La boucle <code>until-do-done</code>	52
10.11.3. La boucle <code>for-do-done</code>	52
I Annexes	53
II Exercices	53
III Powershell	53
IIII Linux	55
III Commandes principales Powershell avec leur équivalent Linux	60

1. Introduction

L'OS (pour Operating System), ou *SE* en français (Système d'Exploitation), est en quelque sorte l'âme de votre ordinateur. Sans cette âme, il ne peut fonctionner.

L'OS est chargé d'assurer la liaison entre vous, utilisateur, et les ressources de l'ordinateur (matérielles et logicielles). Ainsi, lorsqu'un programme désire accéder à une ressource matérielle, il ne lui est pas nécessaire d'envoyer des informations spécifiques au périphérique. Il lui suffit d'envoyer les informations au système d'exploitation, qui se charge de les transmettre au périphérique concerné via son *pilote* (*driver*).

Il existe plusieurs types d'OS. Les trois plus connus sont:

- Microsoft Windows®. C'est l'OS le plus connu et le plus employé dans le monde. Il se décline en plusieurs versions. La plus récente à ce jour est **Windows 11 23H2**.
- Apple MacOS®. Conçu spécifiquement pour les ordinateurs Mac. Deuxième part du marché, les Mac ont l'avantage d'être assez faciles à employer, mais sont beaucoup plus coûteux. Les logiciels sont différents sur Mac ou sur Windows.
- Linux. Cet OS est un système le plus souvent open source et gratuit, mais il semble réservé aux personnes qui connaissent bien l'informatique, même si Linux a tendance à se simplifier. Les versions les plus connues sont **Ubuntu** et **ArchLinux**. Toutefois, nous utilisons sans le savoir le système Linux... sur nos smartphones Android! En effet, **Android** n'est qu'une *surcouche* d'exploitation, qui se base sur Linux pour fonctionner. D'ailleurs, les systèmes d'exploitation Apple (MacOS, Iphone, Ipad) se basent aussi sur un Linux (un Unix en réalité) modifié par Apple.

2. Powershell

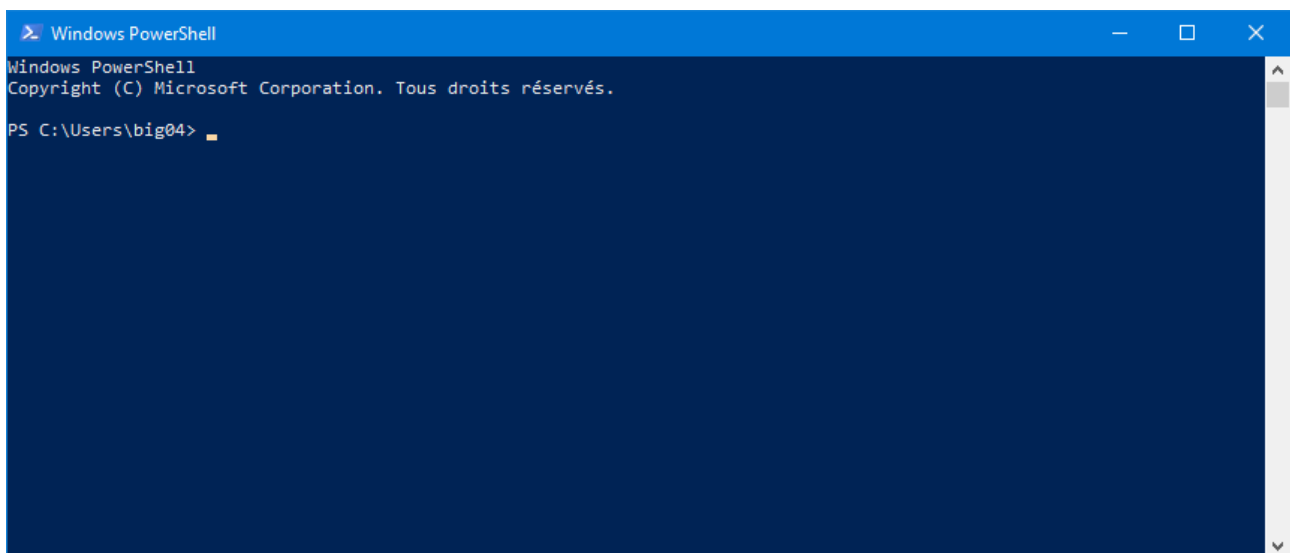
Microsoft Powershell, ou plus simplement la **Powershell**, est un langage de script et un outil de commande en ligne. C'est le successeur de l'ancien *MS-DOS* et l'analogue du **Bash** Linux. La Powershell fournit des outils aux professionnels de l'informatique pour contrôler et automatiser l'administration des systèmes Windows.

Microsoft Powershell est constitué de commandes, nommées *cmdlets*, pouvant traiter avec facilité la gestion des fichiers, du registre, des certificats de sécurité.... La plupart des commandes standard sont reprises, telles que `cd`, `dir`, `md`, `rd` ..., issues du monde DOS, tout comme les équivalents Linux `cd`, `ls`, `mkdir`, `rmdir` Toutefois, Powershell dispose de sa propre notation.




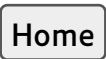

Windows 10 dispose de la version 5.1 de PowerShell. Néanmoins, Microsoft a décidé de passer Powershell en open source depuis la version 6. Cette version est disponible aussi bien sous Windows que sous Linux. Powershell est disponible en version 7.4 à l'adresse <https://github.com/PowerShell> .






Note

Il est préférable de démarrer la Powershell en tant qu'administrateur.



2.1. Les raccourcis clavier

Powershell étant un outil textuel, les raccourcis clavier sont très importants. Par exemple, en utilisant les touches  et , on peut déplacer le curseur clignotant sur la ligne de commande pour la corriger. Si vous appuyez sur la touche  avec les touches fléchées, vous allez vous déplacer mot par mot dans la console. Les touches  et  vous ramèneront au début et à la fin de la ligne.

En cas de faute de frappe, utilisez  pour supprimer le caractère **à gauche** du curseur et la touche  pour supprimer le caractère **à droite**. De plus, sachez qu'en tapant sur la touche , vous supprimerez **toute la ligne**. Les touches  +  vont supprimer

tous les caractères à gauche du curseur, laissant la partie droite intacte. D'une manière similaire, **Ctrl** + **End** va supprimer **tous les caractères à droite du curseur**.

Les autres touches utiles sont : **Insert** pour basculer entre le mode interaction et le mode reffrappe, la touche **Tab** pour compléter la commande en cours d'écriture, les flèches **↑** et **↓** pour afficher les dernières commandes tapées, la touche **F7** pour afficher l'historique des 50 dernières commandes tapées.

Powershell dispose également d'une fonction de recherche dans les commandes précédemment tapées. Tapez # et une portion de texte d'une commande précédemment tapée, puis appuyez autant de fois que nécessaire pour afficher la commande voulue. Tapez sur **Enter** pour valider la commande sélectionnée.

2.2. L'invite de commande interactive

En démarrant Powershell, nous arrivons sur une invite de commande, représentée par =PS C:=. Essayons de taper un calcul.

```
1 2+4
2 >> 6
3 (12+5)*3/4.5
4 >> 11.3333333333333
```

Nous pouvons remarquer que Powershell est une bonne calculatrice. Néanmoins, elle ne sert pas qu'à ça. Par exemple, souhaitons connaître combien de CD on peut stocker sur un DVD.

```
1 4GB/720MB
2 >> 5,68888888888889
```

Un DVD est capable de stocker ±6 CD. En utilisant l'unité arithmétique de stockage, (KB, GB, MB, GB, TB, PB), Powershell comprendra l'unité de mesure et appliquera les calculs correspondants. Il suffit d'accoler l'unité de mesure directement après la valeur numérique, sans espace. De même, Powershell est capable de travailler en octal ou en hexadécimal !

```
1 12+0xCD
2 >> 217
```

Note

Les nombres décimaux sont notés avec le point décimal « . ». En effet, la virgule est destinée à créer des tableaux en Powershell. Faites le test en tapant :

```
1 4,2*2
2 >> 4
3 >> 5
4 >> 4
5 >> 5
```


Note

Les parenthèses ont aussi un rôle spécial avec la Powershell, Elles permettent d'évaluer leur contenu **avant** le reste de la commande. Cela nous servira plus tard.

2.3. Commandes de bases de Powershell

Les commandes de la Powershell sont relativement intuitives. Une **cmdlet** est constituée d'un verbe d'action (*Get, Set, Update, Clear ...*), suivi d'un tiret (-) et de l'objet sur lequel effectuer l'action. Par exemple, la commande `Clear-Host` efface (Clear) l'invite de commande (Host).

La liste des verbes approuvés par Powershell est disponible en tapant `Get-Verb`.

Commençons par la commande la plus utile : `Get-Help`. Cette commande permet d'afficher l'aide sur d'autres commandes. Il s'agit en quelque sorte du manuel des commandes Powershell.

Tapons maintenant : `Get-Help Get-Help`. En lisant la commande, nous souhaitons afficher l'aide (`Get-Help`) sur la commande d'aide (`Get-Help`) ! L'aide de la commande est riche en enseignements, sur l'écriture de la commande en tant que telle, mais aussi sur les paramètres à passer.

Note

Il existe deux alias plus courts qui font appel à `Get-Help` : `help`, qui est identique à l'ancienne commande MS-DOS, et `man`, identique à l'aide sous Linux.

Autre commande de base, `Get-Command`. Cette commande affiche toutes les commandes connues par Powershell. Il existe 3 types de commandes :

- les *cmdlets*, les commandes de base de la Powershell ;
- les *alias*, qui sont des raccourcis vers d'autres fonctions ou cmdlets ;
- les *fonctions*, qui regroupent des cmdlets dans un but précis.

Pour avoir la liste des cmdlets, tapez `Get-Command -CommandType Cmdlet`. La liste des différentes cmdlets est grande. Il est toutefois possible d'utiliser les caractères joker (*) dans la commande `Get-Command` pour filtrer les recherches : `Get-Command *service* -CommandType Cmdlet`.

2.3.1. Utilisation des paramètres

Les paramètres ajoutent des informations aux cmdlets. En prenant l'exemple `Get-Help Get-Help`, nous avons passé en paramètre de `Get-Help` la cmdlet à étudier, *Get-Help*.

Pour en savoir plus sur les paramètres d'une cmdlet, utilisez la commande `Get-Help cmdlet -Parameter *`


Chaque paramètre peut être obligatoire ou optionnel, et se positionne à un endroit précis de la commande (sauf si on utilise les paramètres nommés, indiqué par *named* dans les tableaux).

2.3.1.1. Les paramètres nommés

Les paramètres nommés fonctionnent en paire de clés-valeur. Vous devez spécifier le nom d'un paramètre (commençant toujours par un trait « - »), puis un espace suivi de la valeur à assigner au paramètre.

Exemple :

```
1 Get-ChildItem -Path C:\windows -filter *.exe -Recurse -Name
```

 Powershell

Notez que Powershell autorise une astuce : vous n'êtes pas obligé de noter les paramètres au complet, du moment que vous tapez suffisamment de lettres pour rendre le paramètre non ambigu.

```
1 Get-ChildItem -Pa C:\windows -f *.exe -R -N
```

 Powershell

est l'identique de la formulation ci-dessus.


2.3.1.2. Les paramètres « interrupteurs »

Certains paramètres ne sont pas des paires de clés-valeurs, mais de simples « interrupteurs ». S'ils sont spécifiés, ils activent certaines fonctions. Le paramètre *-Recurse* de la fonction ci-dessus en est un exemple. En anglais, on les appelle des *Switch Parameters*.

2.3.2. Les commandes *Location*

Si vous démarrez la Powershell en administrateur, vous verrez dans le prompt (PS C:\Windows\System32>) le répertoire courant (ici, *system32*). Regardons d'abord comment se repérer dans les répertoires de la Powershell, et naviguer entre eux. Notez que Powershell est capable de naviguer entre les répertoires, mais aussi dans le Registre !

```
1 Get-Command * location * -CommandType Cmdlet
2 >> Get-Location Get-WinHomeLocation Pop-Location
3 >> Push-Location Set-Location Set-WinHomeLocation
```

 Powershell

Faites un *Get-Help* sur chaque commande pour en connaître le contenu.

Get-Location : Cette commande indique le dossier courant.

Set-Location *path* : Cette commande permet de passer dans un autre dossier.


```
1 # Permet de passer au lecteur D:\
2 Set-Location d:\
3 # Permet de passer dans le répertoire c:\users
4 Set-Location C:\users
5 # Permet de remonter d'un répertoire
6 Set-Location ..
```

 Powershell

2.3.3. Les commandes *Item*

Passons aux commandes traitant des Items. Un item peut être un fichier, un dossier ou une clé de registre.

```
1 Get-Command *item* -CommandType Cmdlet
2 >> Clear-Item Clear-ItemProperty Copy-Item
3 >> Copy-ItemProperty Get-ChildItem Get-ControlItem
4 >> Get-Item Get-ItemProperty Get-ItemPropertyValue
5 >> Invoke-Item Move-Item Move-ItemProperty
6 >> New-Item New-ItemProperty Remove-Item
7 >> Remove-ItemProperty Rename-Item Rename-ItemProperty
8 >> Set-Item Set-ItemProperty Show-ControlItem
```

 Powershell

Get-ChildItem [path] : Cette commande permet de lister le contenu d'un répertoire. Sans argument, elle liste le répertoire courant.

```
1 # affiche le contenu du répertoire courant
2 Get-ChildItem
3 # affiche le contenu du répertoire C:\ users
4 Get-ChildItem c:\ users
5 # affiche les fichiers texte du répertoire courant
6 Get-ChildItem *. txt
7 # affiche les exécutables du répertoire C:\ Windows \System32
8 Get-ChildItem -Path C:\ windows \ system32 - Filter *. exe
9 # affiche le contenu des dossiers et des sous-dossiers
10 Get-ChildItem - Recurse
11 # affiche le contenu en lecture seule du répertoire courant .
12 Get-ChildItem - Attributes ReadOnly
13 # affiche le contenu du répertoire courant mais crée une pause à la fin de l'écran
14 Get-ChildItem | more
```

New-Item : Cette commande permet de créer un nouveau dossier ou fichier, mais aussi de nouvelles clés de registre. En cas de création de fichier, nous pouvons écrire du texte via le paramètre *-Value*. Pour créer un dossier, il faut passer le paramètre *-ItemType directory*

```
1 # crée un dossier nommé toto.
2 New-Item toto - ItemType directory
3 # crée un fichier toto.txt dans le répertoire toto avec du texte préinscrit .
4 New-Item -Name toto.txt - ItemType "file" -Path C:\ toto -Value "ceci est un essai "
5 # crée un raccourci nommé toto vers le dossier toto.
6 New-Item -Name toto - ItemType SymbolicLink - Value C:\ toto -Path C:\ users \vco\ desk
```

Copy-Item et Move-Item : Ces commandes sont les commandes de copie et de déplacement (« coupe »). La commande *Copy-Item* copie un élément d'un endroit à un autre, sans effacer la source, au contraire de *Move-Item*. Ces deux commandes peuvent aussi renommer un fichier via le paramètre *-Destination*.

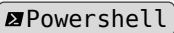
```
1 # copie toto.txt dans les documents .
2 Copy-Item c:\ toto.txt - Destination c:\ users \vco\ documents
3 # copie l'ensemble du dossier folder dans un dossier bkp.
4 Copy-Item c:\ folder - Destination e:\ bkp - Recurse
5 # crée une copie renommée .
6 Copy-Item c:\ folder \toto.txt - Destination c:\ folder \tata.txt
7 # déplace le fichier toto dans les documents .
8 Move-Item c:\ toto.txt - Destination c:\ users \vco\ documents
```

Rename-Item : Cette commande permet de renommer un objet.

```
1 # renomme toto.txt en "mon beau fichier.txt"
2 Rename-Item c:\toto.txt -NewName "mon beau fichier.txt"
```

Remove-Item : Cette commande permet de supprimer un élément (dossier, fichier, clé de registre, alias, fonctions et variables)

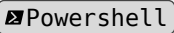
```
1 # supprime le contenu du dossier toto.
2 Remove-Item C:\toto\*.
3 # supprime le dossier et ses sous-dossiers en demandant une confirmation
4 Remove-Item C:\toto -Recurse - Confirm
```



2.3.4. Les commandes *Host*

L'hôte est la fenêtre d'invite de commandes. Voyons comment nous pouvons interagir avec elle.

```
1 Get-Command *host* -Commandtype Cmdlet
2 >> Enter-PSHostProcess Exit-PsHostProcess Get-PSHostProcessInfo
3 >> Get-Host Out-Host Read-Host Write-Host \end{lstlisting}
```

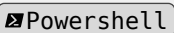


Clear-Host : Vous aurez remarqué que la commande `Clear-Host` n'est pas affichée. Et pour cause, c'est une fonction qui a pour unique but d'effacer l'écran.

Get-Host : Cette commande récupère l'hôte contenant la Powershell (ici, la console ou invite de commandes). Par défaut, `Get-Host` affiche différentes informations, dont la version installée de Powershell, la langue utilisée... Avec le résultat de `Get-Host`, nous pouvons aussi personnaliser les couleurs de l'interface. Nous verrons cela plus loin dans le cours, lors de la création de scripts.

Out-Host : Cette cmdlet est dédiée à l'affichage. Elle envoie le résultat des commandes Powershell dans la console. Cette commande est utilisée par défaut par Powershell. Il n'est donc pas utile de l'appeler explicitement (d'ailleurs, nous ne l'avons jamais fait jusqu'à présent...), sauf si nous souhaitons utiliser des paramètres spécifiques.

```
1 # affiche les dossiers et sous-dossiers du répertoire courant , en
  affichant une "page" à la fois.
2 Get-ChildItem - Recurse | Out-Host - Paging
```



Read-Host et Write-Host : Ces deux commandes permettent respectivement de lire et d'écrire dans la console. `Read-Host` est capable d'afficher un prompt et de masquer la saisie. `Write-Host` dispose de paramètres pour modifier la couleur de texte et de fond de l'écran. Nous utiliserons principalement ces fonctions dans les scripts.

```

1 # Affiche " Entrez votre nom" et attend la saisie utilisateur .
2 Read-Host - Prompt " Entrez votre nom"
3 >> Entrez votre nom :
4 # Idem mais avec un affichage sécurisé .
5 Read-Host - Prompt " Entrez un mot de passe" - AsSecureString
6 # Affiche le message indiqué .
7 Write-Host " Bonjour tout le monde"
8 # Affiche le message sur fond jaune.
9 Write-Host " Bonjour tout le monde" - BackgroundColor Yellow
10 # Affiche le message en bleu.
11 Write-Host " Bonjour tout le monde" - ForegroundColor Blue
12 # le paramètre - NoNewLine empêche le passage à la ligne
13 Write-Host "ligne 1" - NoNewLine #suite en dessous
14 Write-Host "suite de la ligne"
15 >> ligne1 suite de la ligne
16 # affiche chaque élément du tableau avec le séparateur indiqué
17 Write-Host (1 ,2 ,3 ,4 ,5) - Separator ", +1 ="
18 >> 1, +1= 2, +1= 3, +1= 4, +1= 5

```

2.3.5. Les commandes *Computer*

Voyons maintenant les commandes de gestion de l'ordinateur.

```

1 Get-Command * computer * - CommandType Cmdlet
2 >> Add-Computer Checkpoint-Computer Disable-ComputerRestore
3 >> Enable-ComputerRestore Get-ComputerInfo GetComputerRestorePoint
4 >> Remove-Computer Rename-Computer Reset-ComputerMachinePassword
5 >> Restart-Computer Restore-Computer Stop-Computer
6 >> Test-ComputerSecureChannel

```

Les commandes `Add-Computer`, `Remove-Computer` et `Test-ComputerSecureChannel` sont des commandes spéciales pour ajouter ou retirer un ordinateur d'un domaine ou d'un groupe de travail.

Comme leurs noms l'indiquent, `Stop-Computer`, `Restart-Computer` et `Rename-Computer` vont arrêter, redémarrer et renommer l'ordinateur. Pensez à regarder les différents paramètres utilisables pour chaque fonction.

`Get-ComputerInfo` va renvoyer toutes les infos de l'ordinateur.

```

1 # affiche toutes les propriétés utilisables .
2 ( Get-ComputerInfo ) | Get-Member
3 # affiche les informations du bios.
4 Get-ComputerInfo -Property *bios*

```

2.3.5.1. Les commandes liées à la restauration du système

`Enable-ComputerRestore` et `Disable-ComputerRestore` vont respectivement activer ou désactiver les points de restauration du système sur un disque sélectionné. `Get-ComputerRestorePoint` affiche les derniers points de restauration système avec leur n° de séquence, utilisable pour la restauration proprement dite via `Restore-Computer`. Enfin, `Checkpoint-Computer` crée un point de restauration du système. Avec le paramètre `-RestorePointType`, il est pos-

sible d'indiquer le type de point de restauration (installation d'application, suppression d'installation, installation de pilote, changement de paramètres ou annulation d'opération).

```
1 # active la protection du système sur le disque C.
2 Enable-ComputerRestore - Drive C:\
3 # désactive la protection sur le disque C.
4 Disable-ComputerRestore - Drive C:\
5 # crée un point de restauration avec la description indiquée .
6 Checkpoint-Computer - Description " Point de restauration test " - RestorePointType
  MODIFY_SETTINGS
7 # restaure l'ordinateur avec le point 45 (s'il existe ).
8 Restore-Computer - RestorePoint 45
```

2.3.6. Autres commandes utiles

Get-Random : Cette commande renvoie un nombre aléatoire, entre 0 et 2.147.483.647. Avec les paramètres *-Minimum* et *-Maximum*, nous pouvons restreindre la plage. Avec le paramètre *-InputObject* suivi d'une collection d'objets (un tableau de valeurs, par exemple), `Get-Random` sélectionnera aléatoirement une valeur parmi la collection.

Get-PSDrive, New-PSDrive et RemovePSDrive : `Get-PSDrive` permet de lister tous les disques présents sur l'ordinateur, même les disques réseaux et virtuels, ainsi que les clés de registre. De plus, Powershell simule les disques virtuels (nommés fournisseurs), dont les **Alias**, les **Functions**, les **Variables**, les **Environnements** et les **Certificates**.

New-PSDrive : Permet de créer un disque temporaire ou permanent renvoyant vers un endroit (Location). Attention, ces disques ne sont disponibles que via Powershell.

```
1
2 # permet de supprimer un disque temporaire ou permanent créé par New-PSDrive.
3 Remove-PSDrive
4 # affiche tous les disques accessibles via Powershell
5 Get-PSDrive
6 # affiche le disque "alias " et son chemin de localisation .
7 Get-PSDrive alias
8 # crée un lecteur "docs :" pointant vers les Documents .
9 New-PSDrive -Name "Docs" - PSProvider FileSystem -Root "C:\ users\vc0\ Documents "
10 # supprime le lecteur virtuel précédemment créé.
11 Remove-PSDrive -Name "Docs"
12 # fait la même chose .
13 Get-PSDrive "Docs" | Remove-PSDrive
```


2.3.7. Les commandes *Service*

```
1 Get-Command *service* -CommandType Cmdlet
2 >> Get-Service New-Service New-WebServiceProxy
3 >> Restart-Service Resume-Service Set-Service
4 >> Start-Service Stop-Service Suspend-Service
```

Ces différentes fonctions servent à gérer les services.

2.3.8. Les commandes *User*

```
1 Get-Command *User* -CommandType Cmdlet
2 >> Disable-LocalUser Enable-LocalUser Get-LocalUser
3 >> New-LocalUser Remove-LocalUser Set-LocalUser
```

 Powershell

2.4. Les alias

Nous l'avons vu, les cmdlets sont constituées d'une convention verbe-objet, certes très pratique pour comprendre le but de la cmdlet, mais parfois trop long à écrire quand on veut exécuter plusieurs actions rapidement. C'est pourquoi Powershell dispose d'*alias*, c.-à-d. des commandes renommées et raccourcies.

Ces alias sont souvent des commandes « historiques » issues du monde DOS et Linux. Par exemple, `Get-ChildItem` possède l'alias `dir` (DOS) et `ls` (Linux). Les alias ont deux buts importants dans la Powershell:

historique : les utilisateurs peu familiers à la Powershell peuvent utiliser les commandes DOS ou Linux standards ;

vitesse : les cmdlets avec des alias plus courts sont plus rapides et plus faciles à retenir.

Les alias sont stockés dans le disque virtuel *alias*. Listons ces alias.

```
1 Get-ChildItem alias : # ou Get-Alias
2 >> Alias % -> ForEach-Object
3 >> Alias ? -> Where-Object
4 >> Alias ac -> Add-Content
5 >> Alias asnp -> Add-PSSnapin
6 >> Alias cat -> Get-Content
7 >> Alias cd -> Set-Location
8 >> ...
```

 Powershell

La commande `Get-Alias` permet de faire la même liste. Désormais, nous privilégierons l'utilisation des alias dans l'invite de commandes pour faire plus court. Néanmoins, il est préférable d'utiliser les noms d'origine lors de la création de scripts.

2.4.1. Créer un alias

La création d'un alias se fait via la commande `New-Alias`. Les alias créés avec cette cmdlet seront perdus à la sortie de Powershell. Pour conserver ses jeux d'alias, il faudra les exporter (via `Export-Alias`) et les importer à chaque fois que nous en aurons besoin (via `Import-Alias`).

Il est tout à fait possible de créer un alias pour une application comme pour une cmdlet.

```

1 # crée l'alias "edit" pour notepad .
2 New-Alias -Name "edit" - Value " notepad .exe"
3 # version courte
4 New-alias "edit" " notepad .exe"
5 # exporte les alias
6 Export-Alias -Path " alias.csv"
7 # exporte les alias si le fichier n'existe pas.
8 Export-Alias -Path " alias.csv" -NoClobber
9 # ajoute les alias au fichier script alias .ps1.
10 Export-Alias "alias .ps1" - Append -As Script
11 # importe les alias .
12 Import-Alias -Path "alias.csv"

```

 Powershell

Il est possible de charger automatiquement une série d'alias quand l'utilisateur démarre la Powershell. Nous verrons cela plus tard.

2.5. Le Pipeline

Dans la Powershell, un pipeline est un « tuyau » connectant deux commandes. Le résultat d'une commande est envoyée pour traitement dans une autre commande. Pour connecter deux commandes entre elles, il faut les séparer par la barre verticale (|). Par exemple, la commande suivante : `Get-ChildItem -Path C:\windows\system32 | Out-Host -Paging`.

On peut le traduire ainsi: le résultat de la commande `Get-ChildItem` va lister le contenu du répertoire *system32*. Ce résultat, au lieu d'être affiché à l'écran directement, va passer par la cmdlet `Out-Host` qui va afficher le résultat à l'écran en créant des « pages ».

Une des applications les plus courantes du pipe est l'utilisation de la cmdlet `Get-Member`. Chaque commande lancée dans la console produit un résultat affiché. Néanmoins, le résultat réel est un objet beaucoup plus complexe et le résultat en console ne représente qu'une fraction de cet objet. Pour connaître toutes les propriétés de l'objet et les utiliser, il faut passer l'objet obtenu par la cmdlet à `Get-Member`, qui va afficher ses propriétés.

Il est tout aussi faisable de lier un ensemble de commandes par les pipes: `Get-ChildItem -Path c:\windows\system32 | Sort-Object Length | Select-Object Name, Length | ConvertTo-HTML | Out-File report.htm`.

Cette commande va lister le répertoire *system32* en triant les résultats par taille de fichiers, afficher les noms et leur taille, convertir le résultat en HTML pour les écrire dans le fichier `report.htm`.

Avec les alias, nous aurions pu écrire aussi `gci c:\windows\system32\ | sort Length | select Name,Length | ConvertTo-HTML | Out-File report.htm`.

3. Conception de scripts

Maintenant que nous nous sommes familiarisés avec les commandes de base, nous pouvons passer à l'automatisation de tâches plus complexes grâce à l'écriture de scripts.

Un script Powershell est un simple fichier texte portant l'extension « .ps1 ». Il peut donc être modifié par un éditeur de texte quelconque. Néanmoins, il existe un outil puissant pour nous aider à la conception de scripts: **VS Code** avec l'extension *Powershell*.

Note

Attention, par défaut, il existe une sécurité qui empêche l'exécution des scripts. Tapez `Get-ExecutionPolicy` pour connaître le niveau de sécurité (*Restricted* par défaut). Changez ce niveau de sécurité via la commande `Set-ExecutionPolicy RemoteSigned`.

Avec la conception de scripts, il sera plus facile d'automatiser une suite de commandes, mais aussi nous avons désormais la possibilité de concevoir de petits programmes de gestion de l'ordinateur. Nous verrons plus tard quelques notions avancées sur les scripts.

3.1. Les variables

Il est parfois nécessaire de combiner plusieurs commandes Powershell pour obtenir le résultat souhaité. Mais pour combiner efficacement ces commandes, il faut être capable de stocker ou récupérer les résultats des commandes précédentes pour les passer aux autres. C'est justement le rôle des *variables*.

3.1.1. Création de variables

```
1 # Calculation of VAT
2 # Create variables and assign to values
3 $amount = 120
4 $VAT = 0.21
5
6 # Calculate
7 $result = $amount * (1+$VAT)
8
9 # Display the result
10 $text = "Si le montant HT est de $amount €, le montant TTC sera de $result €"
11 $text
```

Pour créer une variable, il suffit de la *nommer* en précédant son nom d'un \$, puis de lui assigner une valeur. Notez qu'il n'est pas nécessaire de *typer* les variables, Powershell déduit leur type en fonction de leur assignation. Toutefois, cette propriété nommée *inférence de type* peut poser des problèmes:

```
1 $a=12 # la variable a est un entier (Integer)
2 Write-Host $a/2
3 $a="Toto" # la variable a est maintenant du texte (String)
4 Write-Host $a/2
5 >> Error...
```

Il est toutefois possible de forcer le typage des variables (voir listing suivant), même si en pratique on ne le fera que pour augmenter la sécurité des scripts et éviter certaines erreurs.

```
1 # Calculation of VAT
2 # Create variables and assign to values
3 [Int]$amount = Read-Host -Prompt 'Entrez la valeur HT' # Forcing integer value
4 [Double]$VAT = Read-Host -Prompt 'Entrez la TVA (%)' # Forcing double value
5 $VAT /= 100 #revise to correct VAT
6
7 # Calculate
8 $result = $amount * (1+$VAT)
9
10 # Display the result
11 $text = "Si le montant HT est de $amount €, le montant TTC sera de $result €"
12 $text
```

Note

Il est important de choisir judicieusement le nom de vos variables. Dans l'idéal, il faut que le nom de celle-ci définisse clairement sa fonction. Évitez donc de mettre des noms trop petits ou trop générique (\$a~, ~\$b ...). Évitez aussi de mettre des noms trop longs!

Il est toutefois possible de définir des variables avec des espaces dans leur nom, même si cette pratique est déconseillée. Pour ce faire, il suffit de mettre le nom de la variable entre accolades: \${mon petit nom de variable bien trop long}.

3.1.2. Vérifier l'existence d'une variable

En utilisant la cmdlet `Test-Path`, nous pouvons vérifier si un fichier existe. Rappelons aussi que Powershell dispose de disques virtuels, dont un qui « stocke » toutes les variables. En combinant les deux, nous avons donc la possibilité de tester l'existence d'une variable!
Ex: `Test-Path variable:\psversiontable`.

3.1.3. Les cmdlets de gestion des variables

Pour gérer les variables, Powershell fournit des cmdlets spécifiques.

```
1 gcm *variable* -Com Cmdlet
2 >> Clear-Variable Get-Variable New-Variable
3 >> Remove-Variable Set-Variable
```

Clear-Variable : Efface le contenu de la variable, mais sans la supprimer. La valeur contenue dans la variable sera `$null`. Si la variable est explicitement typée, elle garde le type de la valeur à stocker. Cela revient à écrire `$a=$null`.

Get-Variable : Récupère la variable en tant qu'objet, et non la valeur contenue dans la variable.

New-Variable : Crée une nouvelle variable et permet d'initialiser certaines options spécifiques (telles que la description, la lecture seule ou la constante).

Remove-Variable : Supprime la variable et son contenu, pour autant que la variable ne soit pas une constante ou que ce ne soit pas une variable système. Cela revient à écrire `del variable:\a`.

Set-Variable : Réinitialise le contenu de la variable ou les options d'une variable existante. Crée une variable si elle n'existe pas. Cela revient à écrire `$a=12`.

3.1.4. Variables en lecture seule et constante

Par définition, une variable est une zone mémoire qui peut évoluer au fil du script. Mais dans certains cas, il est nécessaire de créer des variables qui conservent leur valeur. Nous les appelons *constantes*. Toutefois, Powershell ne fait pas la différence entre une variable et une constante. Par contre, le système fournit la possibilité de protéger une variable en écriture.

```
1  # Crée une variable avec une description et une protection en écriture
2  New-Variable test -Value 100 -Description "Test variable with write-protection" -Option
   ReadOnly
3
4  "La variable test vaut $test"
5
6  # Réécriture de la variable test
7  "Tentative de réécriture"
8  $test = 200 # Une erreur devrait s'afficher.
9
10 #Tentative de suppression
11 "Suppression de la variable"
12 del variable:\test
13
14 # on peut toutefois détruire la variable en forçant sa suppression.
15 "Suppression de la variable forçant sa suppression"
16 del variable:\test -Force
17
18 $test = 200 #on peut recréer la variable avec une nouvelle valeur. Cette variable perd
   sa protection.
19
20 "La variable test vaut maintenant $test"
21
22 # Crée une constante, impossible à modifier et à supprimer, existant pendant toute la
   durée du script.
23 New-Variable const -Value 100 -Description "Constante" -Option Constant
24
25 "La constante vaut $const"
26
27 # Tentative de suppression
28 "Tentative de suppression"
29 Remove-Variable const
30
31 # Tentative en forçant
32 "Tentative de suppression forcée"
33 Remove-Variable const -Force
```

Note

Il est possible d'ajouter une description aux variables. Toutefois, ces descriptions n'apparaissent pas par défaut, sauf en les appelant via `Get-Member`.

3.1.5. Les variables automatiques

Powershell possède une série de variables automatiques utilisées à des fins internes. Ces variables sont disponibles dès le lancement de Powershell. Affichez la liste de ces variables pour en connaître leur utilité:

```
gci variable: | sort Name | Format-Table Name,Description -
Autosize -Wrap
```

3.1.6. Typage des variables

Nous l'avons vu précédemment, il n'est pas nécessaire de définir le type de contenu d'une variable. En fonction du contenu de celle-ci, Powershell définit de lui-même le bon type de données. Pour connaître le type d'une variable, il suffit d'appeler la méthode `GetType` sur un objet. Ex: `$a.GetType`. **Attention**, `GetType` n'est pas une cmdlet.

```
1 "Le type de 12 est " + (12).GetType().Name  
2 "Le type de 1000000000000000 est " + (1000000000000000).GetType().Name  
3 "Le type de 12.5 est " + (12.5).GetType().Name  
4 "Le type de 12d est " + (12d).GetType().Name  
5 "Le type de 'H' est " + ("H").GetType().Name  
6 "La date du jour est " + (Get-Date).GetType().Name
```

 PowerShell

Ce processus est nommé le *typage faible* et, même s'il est facile à utiliser, il présente certains risques, car une variable faiblement typée peut accepter *n'importe* quel type de données. De plus, Powershell peut se tromper en définissant le type de données.

En pratique, il existe deux raisons pour créer un typage fort, c.-à-d. spécifier explicitement le type de donnée d'une variable :

- **La sécurité.** En créant un typage fort, la variable conserve ce type de données pour votre variable. En cas de mauvais type (par exemple, en tentant de stocker du texte dans un Int32), le système renverra une erreur.
- **La spécialisation.** Lors d'un typage faible, Powershell n'utilise que des types génériques. Il est donc parfois utile de stocker les variables dans un format plus spécialisé.

3.2. Les conditions

Les conditions sont des éléments essentiels dans l'écriture des scripts. Elles peuvent évaluer des situations et apporter des actions appropriées. Une condition est en réalité une sorte de question dont la réponse ne peut être que *true* (vrai) ou *false* (faux).

3.2.1. Les opérateurs de comparaison

Avant de savoir comment poser une condition dans Powershell, intéressons-nous d'abord aux opérateurs de comparaison.

Opérateur	Description	Exemple	Résultat
-eq, -ceq, -ieq	Égalité	10 -eq 15	false
-ne, -cne, -ine	Inégalité	10 -ne 15	true
-gt, -cgt, -igt	Plus grand	10 -gt 15	false
-ge, -cge, -ige	Plus grand ou égal	10 -ge 15	false
-lt, -clt, -ilt	Plus petit	10 -lt 15	true
-le, -cle, -ile	Plus petit ou égal	10 -le 15	true
-contains, -ccontains, -icontains	Contient	1,2,3 -contains 1	true
-notcontains, -cnotcontains, -inotcontains	Ne contient pas	1,2,3 -notcontains 1	false

Note

Chaque opérateur de comparaison est décliné en 3 variantes. La première est utile pour comparer les nombres. Les deux autres servent à comparer du texte. Les variantes commençant par « c » sont dites *case-sensitive* (sensible à la casse, donc fait la différence entre les minuscules et les majuscules); les variantes commençant par « i » sont dites *case-insensitive* (insensible à la casse, donc ne fait pas la différence entre minuscules et majuscules).

3.2.2. Les opérateurs logiques

Dans certains cas, il faut combiner plusieurs conditions pour effectuer une action. Par exemple, `($age -ge 18) -and ($sex -ieq "m")` indique qu'il faut être majeur **et** de sexe masculin. Il existe quatre opérateurs logiques :

-and : **Toutes** les conditions doivent être **vraies** pour obtenir un résultat **vrai**.

-or : **Au moins une** condition doit être **vraie** pour obtenir un résultat **vrai**.

-xor : **Une seule** condition doit être **vraie** pour obtenir un résultat **vrai**.

-not : **Inverse** le résultat.

3.2.3. If-ElseIf-Else

Pour bien comprendre le fonctionnement de la structure conditionnelle, analysons le script suivant :

```

1 # Exemple de condition If-ElseIf-Else
2 $name = Read-Host -Prompt "Indiquez votre nom"
3 $age = Read-Host -Prompt "Indiquez votre âge"
4 $sex = Read-Host -Prompt "Indiquez votre sexe"
5
6 Write-Host "Bonjour $name !"
7
8 # Condition
9 If ($sex -ieq "m") {
10     Write-Host "Tu es un homme, et tu as $age ans"
11 }
12 ElseIf ($sex -ieq "f"){
13     Write-Host "Tu es une femme, et tu as $age ans"
14 }
15 Else {
16     Write-Host "Tu es quelque chose, mais quoi... En tous cas, tu as $age ans !"
17 }

```

Une structure conditionnelle s'écrit avec `if`, suivi de la comparaison entre parenthèses, puis d'un couple d'accolades. Si la condition est vraie, les lignes de codes inscrites entre les accolades sont exécutées. Sinon elles sont ignorées.

3.2.4. Switch

Analysons maintenant le code suivant :

```

1 # Exemple de condition If-ElseIf-Else
2 $name = Read-Host -Prompt "Indiquez votre nom"
3 $age = Read-Host -Prompt "Indiquez votre âge"
4 $sex = Read-Host -Prompt "Indiquez votre sexe"
5
6 Clear-Host
7 Write-Host "Bonjour $name !"
8
9 # Condition
10 If ($sex -ieq "m") {
11     Write-Host "Tu es un homme, et tu as $age ans"
12 }
13 ElseIf ($sex -ieq "f"){
14     Write-Host "Tu es une femme, et tu as $age ans"
15 }
16 Else {
17     Write-Host "Tu es quelque chose, mais quoi... En tous cas, tu as $age ans !"
18 }
19
20 If ($age -lt 18) { "Tu es encore un bébé !" }
21 ElseIf (($age -ge 18) -and ($age -lt 25)) { "Tu es un jeune adulte" }
22 ElseIf (($age -ge 25) -and ($age -lt 40)) { "Tu es un adulte" }
23 Else { "Tu es vieux" }

```

Nous voyons dans ce listing une succession de `ElseIf` pour commenter l'âge de la personne. Ce procédé fonctionne, mais il est compliqué à lire et est de plus tout aussi compliqué à modifier. Il existe une écriture plus simple et plus efficace: le `switch`. C'est une structure conditionnelle qui va vérifier la valeur d'une variable.

```
1 # Exemple de condition If-ElseIf-Else Powershell
2 $name = Read-Host -Prompt "Indiquez votre nom"
3 $age = Read-Host -Prompt "Indiquez votre âge"
4 $sex = Read-Host -Prompt "Indiquez votre sexe"
5
6 Clear-Host
7 Write-Host "Bonjour $name !"
8
9 # Condition
10 If ($sex -ieq "m") {
11     Write-Host "Tu es un homme, et tu as $age ans"
12 }
13 ElseIf ($sex -ieq "f"){
14     Write-Host "Tu es une femme, et tu as $age ans"
15 }
16 Else {
17     Write-Host "Tu es quelque chose, mais quoi... En tous cas, tu as $age ans !"
18 }
19
20 Switch ($age) {
21     { $_ -lt 18 } { "Tu es encore un bébé !" }
22     18 { "Tu as atteint la majorité" }
23     { ($_ -gt 18) -and ($_ -lt 25) } { "Tu es un jeune adulte" }
24     { ($_ -ge 25) -and ($age -lt 40)} { "Tu es un adulte" }
25     Default { "Tu es vieux" }
26 }
```

3.3. Les boucles

Une boucle permet de répéter plusieurs fois des lignes de code. C'est un des concepts essentiels de la programmation, dans n'importe quel langage.

Il existe trois types de boucles: les *finies*, les *indéfinies* et les *boucles sur les collections*.

3.3.1. ForEach et ForEach-Object

Certaines commandes peuvent retourner plusieurs résultats. Si on veut traiter tous les résultats un par un, sans en omettre, nous allons passer par la structure `ForEach`.

Note

Attention, il existe *deux* « commandes » `ForEach`. La première est un alias de la cmdlet `Foreach-Object`, utilisable principalement avec les pipelines. La seconde est l'écriture `C#`, plus adaptée aux scripts.

```

1 # Écriture en pipeline
2 Get-ChildItem C:\ | ForEach-Object {$_.name}
3 # Écriture en script
4 foreach ($element in Get-ChildItem C:\) {$element.name}

```

 Powershell

3.3.2. Les boucles indéfinies

Ici aussi, il existe deux types de boucles indéfinies: la boucle `Do~` et la boucle `~While`. Ces deux boucles vérifient à chaque tour si la condition d'arrêt est atteinte. Il faut toutefois faire attention! Si la condition est mal paramétrée, la boucle peut ne jamais s'arrêter et devenir une boucle infinie!

La différence entre les boucles `Do..While` et `While` se situe à l'emplacement de la condition d'arrêt. Dans la boucle `Do..While`, la condition est placée **à la fin** de la boucle, exécutant donc le code au cœur de la boucle au moins une fois. Dans la boucle `While`, la condition d'arrêt se situe **en début** de boucle. Si la condition est déjà atteinte, le code ne sera même pas exécuté.

```

1 # Boucle Do
2 New-Variable -Name a
3 do {
4     $a = Read-Host -Prompt "Entrez une valeur entre 0 et 10" #Ce code sera exécuté au
      moins une fois
5 } while (-not (0..10 -contains $a))
6
7 #Boucle While
8 $cpt = 1
9 while ($cpt -le 10) {
10     "Ceci est la ligne $cpt"
11     $cpt += 1 # Ne pas oublier d'incrémenter le compteur, sinon on obtient une
      boucle infinie!
12 }

```

 Powershell


3.3.3. La boucle finie

La boucle `For` est un type particulier de boucle, avec une écriture tout aussi particulière. En effet, la condition de départ est triple: nous avons en premier l'initialisation de la variable, ensuite son critère de continuité et enfin la valeur d'incrémementation.

```

1 # Boucle For
2 for ($freq = 1000; $freq -le 4000; $freq += 300){
3     [System.Console]::Beep($freq,100)
4 }

```

 Powershell

4. Introduction à Linux avec ArchLinux

Linux est un système d'exploitation, à l'instar de Windows ou de MacOS. C'est en quelque sorte l'âme de l'ordinateur, qui permet de faire fonctionner le matériel. Néanmoins, il est moins, il est moins connu que ses deux concurrents.

Toutefois, vous utilisez sans le savoir le système Linux... sur votre smartphone (ou votre tablette) Android! En effet, Android n'est qu'une « surcouche » d'exploitation, qui se base sur un noyau Linux pour fonctionner. D'ailleurs, les systèmes d'exploitation Apple (MacOs, Iphone, Ipad), se basent, eux aussi sur un système Linux modifié par Apple. Linux est aussi très présent dans le monde des serveurs, notamment sur les serveurs web.

Mais alors, pourquoi Linux n'est-il pas aussi connu et utilisé ? Nous pouvons déjà donner un élément de réponse : Windows et MacOS sont respectivement maintenus par leur société mère, Microsoft et Apple, qui paient des équipes de développement pour faire connaître et améliorer leurs produits.

Linux, quant à lui, dispose d'une multiplicité de variantes, et qui sont souvent maintenues par des développeurs bénévoles. Linux est open source, ce qui signifie qu'il peut être gratuit (et c'est souvent le cas !) mais attention, open source ne veut pas dire *gratuit*. Il existe donc plusieurs distributions Linux différentes (que l'on nomme GNU/Linux, pour être précis), ce qui rend les choses plus compliquées.

Néanmoins, depuis presque 30 ans, des entreprises gravitent dans l'écosystème Linux, telles que Red Hat, Mandriva, Debian, ou encore Canonical, qui fait connaître une version de Linux spécifique, Ubuntu, qui lui-même est dérivé en plusieurs *saveurs*.

4.1. Pourquoi Linux ?

Le but de ce chapitre n'est pas de vous faire changer de l'environnement Windows à l'environnement Linux. Par contre, Linux peut vous permettre de faire fonctionner un ancien ordinateur et le remettre au goût du jour !

Linux est réputé pour sa sécurité, sa stabilité et ses mises à jour plus fréquentes que Windows. De plus, Linux, contrairement à ses concurrents, n'installe que ce qui est nécessaire à votre ordinateur.

4.2. Pourquoi différentes *distributions* ?

Comme dit plus haut, Linux existe en différentes distributions. Mais expliquons d'abord le principe de distribution : c'est un ensemble préétabli de logiciels, installés de base avec le système d'exploitation.

Le choix d'une distribution n'est pas si anodin que ça : même si globalement, toutes les distributions se valent, en proposant les mêmes programmes et fonctionnalités, chaque distribution diffère dans sa capacité d'évolution, sa politique d'utilisation des logiciels autres qu'open source, sa fréquence de mise à jour, ou tout simplement ne cible pas le même public. En fonction de la puissance de votre ordinateur et de vos besoins, vous allez choisir la meilleure distribution.

4.3. Pourquoi des interfaces graphiques différentes?

Avant de commencer, définissons l'interface graphique. Quand vous démarrez votre ordinateur Windows, vous arrivez sur un bureau, avec un bouton Démarrer, l'accès à vos fichiers

via un explorateur... Tout ce que vous voyez à l'écran constitue l'interface graphique. De même, pour MacOS, vous avez également cet environnement de bureau, avec le *Finder*.

Ces éléments sont propres aux systèmes d'exploitation, et leur utilisation se veut simple, claire et efficace. De la même manière, les tablettes et smartphones Android ont eux aussi un environnement de bureau!

Autre point propre à l'univers Linux (avantage ou inconvénient, le débat est ouvert), il existe plusieurs interfaces graphiques (ou environnements de bureau) différentes. Là aussi, ce phénomène est dû à la philosophie de l'open source, et aux choix des différentes équipes de développement.

Il existe deux grandes familles d'environnement de bureau, basés sur des « bibliothèques graphiques ». Une bibliothèque graphique est un ensemble de procédures et d'outils de programmation d'un environnement de bureau.

Donc, puisqu'il existe deux grandes bibliothèques graphiques, il existe donc deux grands environnements de bureau en Linux: *Gnome* et *KDE*. Ces environnements existent depuis près de 30 ans et ont évolué. Mais l'univers du libre permet de faire beaucoup de choses! Il suffit que l'évolution de ces deux géants fasse quelques mécontents, et par la magie de l'open source, des équipes peuvent librement récupérer (et parfois faire ressusciter) des anciennes versions, en les modifiant quelque peu. D'autres équipes ont aussi récupérés les bibliothèques graphiques de base pour faire d'autres environnements de bureau différents, et souvent plus léger. Car c'est un fait: Gnome et KDE sont des poids lourds, dans tous les sens du terme! De ces dérivations (dans le jargon Linux, on appelle cela des « *forks* ») sont apparus des environnements de bureau tels que DDE (propre à Linux Deepin), Cinnamon, Mate, XFCE, LXDE, Budgie ou encore Pantheon (propre à Elementary OS). Vous avez l'embarras du choix!

4.4. Pourquoi ArchLinux?

ArchLinux (<https://archlinux.org>) : C'est une distribution Linux ayant la réputation d'être complexe à installer. C'est toutefois une architecture « **KISS** » (Keep It Simple and Stupid), à savoir le plus simple et le plus bête qu'il soit. Elle dispose désormais d'un outil nommé *archinstall* qui permet d'installer ce système d'exploitation assez facilement.

Manjaro (<https://manjaro.org>) : Distribution fille d'ArchLinux, la plus connue. Manjaro est relativement simple d'utilisation, mais il fonctionne avec des outils différents de ceux de Debian, la distribution la plus connue. Toutefois, il est une excellente alternative à l'architecture Ubuntu. Manjaro existe dans différentes « saveurs »: Gnome, KDE, XFCE sont bien évidemment de la partie, mais il existe aussi Budgie, Cinnamon, Mate et encore bien d'autres!

EndeavourOS (<https://endeavouros>) : C'est une distribution « sœur » de Manjaro, plus facile à installer et à utiliser au quotidien. Son approche est plus respectueuse de ArchLinux, contrairement à Manjaro qui s'oriente plus vers le grand public.

4.4.0.0.0.1. Que faut-il télécharger?

Pour ce cours, il fallait bien choisir une distribution. Notre choix s'est porté sur ArchLinux avec l'interface Gnome. Chaque distribution se télécharge sous la forme d'un fichier *iso*. Un fichier iso est une image d'un DVD à graver sur un média physique pour l'utiliser, même

si les ordinateurs récents peuvent utiliser l'image iso directement en montant un DVD virtuel.

4.5. Tester ArchLinux sans risques

Il y a plusieurs façons de tester une distribution Linux sans risques, mais chacune d'entre elles demande un peu de préparation. Rien de très compliqué toutefois, tout sera expliqué ici.

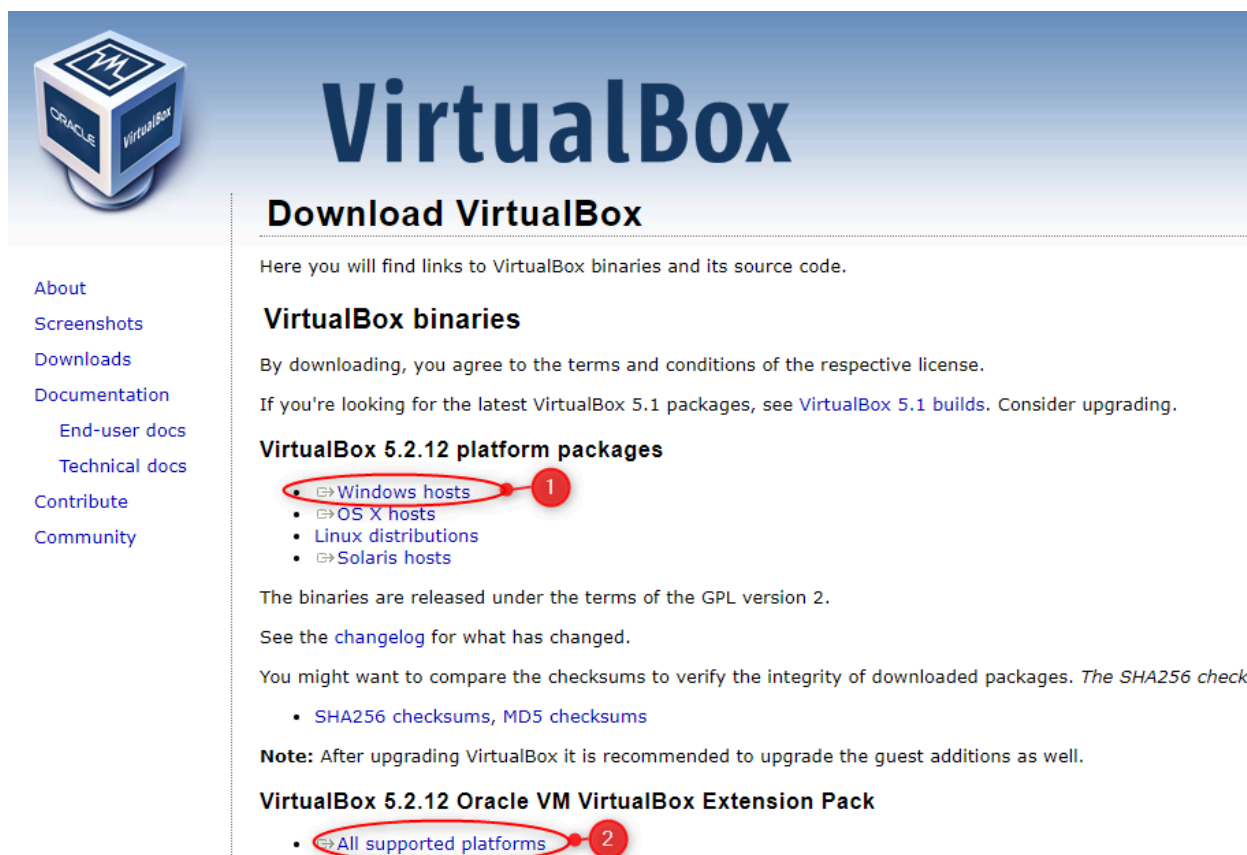
4.5.1. La machine virtuelle

La première méthode sans risques est l'utilisation d'une machine virtuelle. Une machine virtuelle est un logiciel simulant une machine physique. Elle permet d'installer et d'utiliser un système d'exploitation comme une machine réelle. La machine virtuelle est isolée de la machine physique. En contrepartie, l'utilisation d'une machine virtuelle demande un ordinateur relativement puissant, car il doit faire fonctionner la machine hôte *et* la machine virtuelle en même temps. De plus, les performances du système Linux ne seront pas les mêmes qu'en le testant directement sur la machine sur laquelle on souhaite l'installer.

4.5.1.1. Installer une machine virtuelle

Nous allons utiliser *Oracle VM VirtualBox* pour créer des machines virtuelles. Téléchargez sur le site <https://www.virtualbox.org>:

1. Virtualbox proprement dit, version Windows;
2. Les extensions, pour activer le support des clés USB notamment (optionnel, mais vivement conseillé).



VirtualBox

Download VirtualBox

Here you will find links to VirtualBox binaries and its source code.

VirtualBox binaries

By downloading, you agree to the terms and conditions of the respective license.

If you're looking for the latest VirtualBox 5.1 packages, see [VirtualBox 5.1 builds](#). Consider upgrading.

VirtualBox 5.2.12 platform packages

- [Windows hosts](#) **1**
- [OS X hosts](#)
- [Linux distributions](#)
- [Solaris hosts](#)

The binaries are released under the terms of the GPL version 2.

See the [changelog](#) for what has changed.

You might want to compare the checksums to verify the integrity of downloaded packages. *The SHA256 checks*

- [SHA256 checksums](#), [MD5 checksums](#)

Note: After upgrading VirtualBox it is recommended to upgrade the guest additions as well.

VirtualBox 5.2.12 Oracle VM VirtualBox Extension Pack

- [All supported platforms](#) **2**

Installez ensuite le logiciel comme à l'accoutumée. Une fois Oracle VM Virtualbox installé, *ne le démarrez pas*. Cliquez sur les extensions (dont l'icône sera un cube vert) pour les installer.

4.5.1.2. Installer ArchLinux dans une machine virtuelle

Commençons par créer une machine virtuelle dans Oracle VM Virtualbox. Au lancement de l'interface, repérez le bouton **Nouvelle**. Un assistant de création en plusieurs étapes va apparaître:

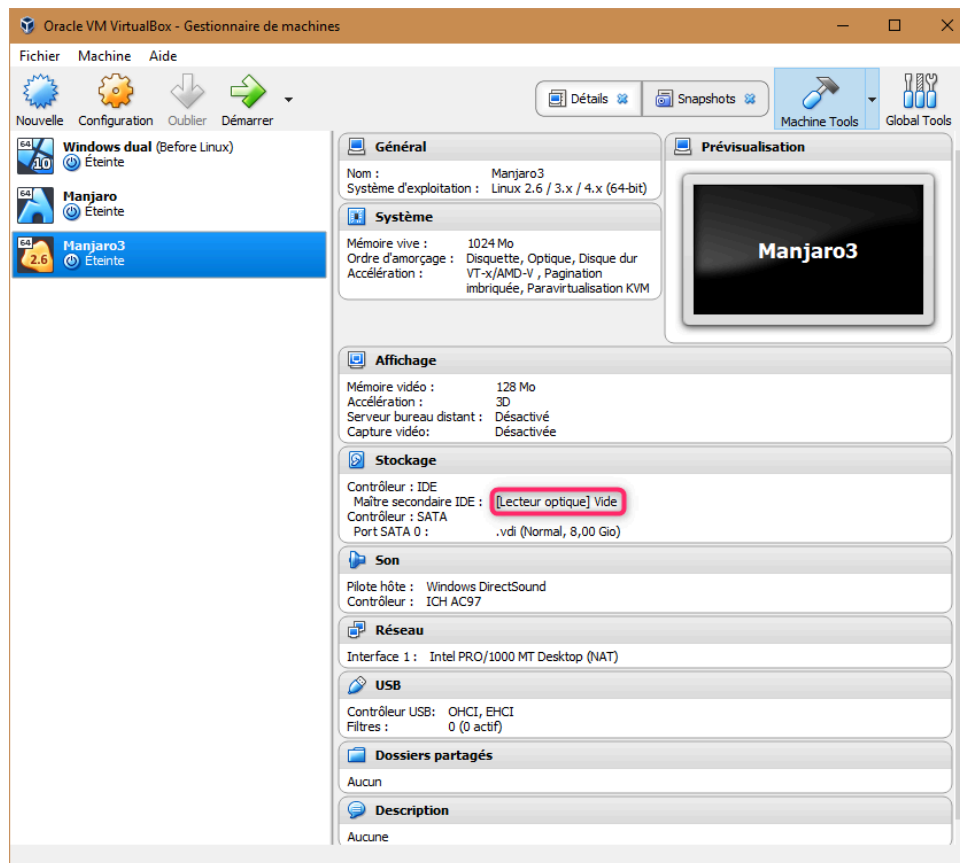
Nom : Indiquez le nom de votre machine virtuelle (*Linux*, par exemple). Notez que le type, la version et l'icône changeront automatiquement selon le nom donné. Mais il vaut mieux que les informations soient *Type Linux* et *Version Linux 4.x* ou *ArchLinux*. Passez à l'étape suivante;

Mémoire : Réglez le sélecteur de mémoire à une quantité raisonnable. Il faut que le sélecteur reste dans la zone verte, sinon la machine physique deviendra instable. Essayez d'avoir au minimum 4 Go de mémoire;

Disque dur : Dans l'écran suivant, nous allons régler le comportement du disque dur virtuel. Ce disque dur sera en réalité un fichier autonome dans Windows, qui ne modifiera en rien le système d'exploitation hôte. Laissez l'option sur *Créer un disque dur virtuel maintenant*, puis cliquez sur **Créer**.

- Laissez le type de fichier de disque dur à *VDI*;
- Le stockage sur disque dur physique peut se faire de deux façons: en stockage *Dynamiquement alloué*, l'espace sur le disque dur grandira en fonction de l'utilisation que vous en faites, pour une vitesse moins élevée (mais ce n'est pas flagrant...). En *Taille fixe*, la vitesse est plus élevée, mais ça bloque la taille maximale du disque créé. Préférez le stockage *Dynamiquement alloué*.
- indiquez maintenant le nom du disque dur et sa taille. Pour des tests, une taille de 8 à 40 Go est amplement suffisante. Cliquez sur le bouton **Créer** pour générer le disque virtuel.

Dès que la machine virtuelle est créée, l'interface affichera le détail de la machine. Cliquez sur *Affichage* pour pousser la mémoire à 128 Mo. Cliquez ensuite sur *Lecteur Optique* pour y insérer le disque virtuel au format iso. La machine virtuelle est prête à être lancée! Cliquez sur le bouton **Démarrer** pour lancer le système et installer Linux.



4.5.2. Live CD / Live USB

La seconde méthode est l'utilisation d'un live cd, live dvd ou live usb. Cette méthode permet de tester Linux directement sur la machine physique, sans pour autant l'installer. Il suffit d'insérer le lecteur au démarrage de l'ordinateur pour démarrer sur Linux, et de retirer le lecteur une fois l'ordinateur éteint pour retrouver son Windows tel qu'il était.

L'inconvénient mineur de cette solution est la création d'une clé usb ou la gravure d'un cd. Le système Linux sera aussi dépendant de la vitesse de lecture du périphérique (préférez une clé USB 3 pour avoir un système plus vélocé, mais il ne sert à rien d'utiliser une clé usb 3 si votre ordinateur n'a pas de prise ad hoc, ce qui est probablement le cas si vous souhaitez réutiliser un vieil ordinateur). L'autre petit inconvénient est l'impossibilité d'installer des logiciels et de stocker des documents. Contrairement à la machine virtuelle, on a toutefois la possibilité d'accéder à ses informations Windows (pratique si le Windows en question ne démarre plus!).

4.5.2.1. Créer un live USB

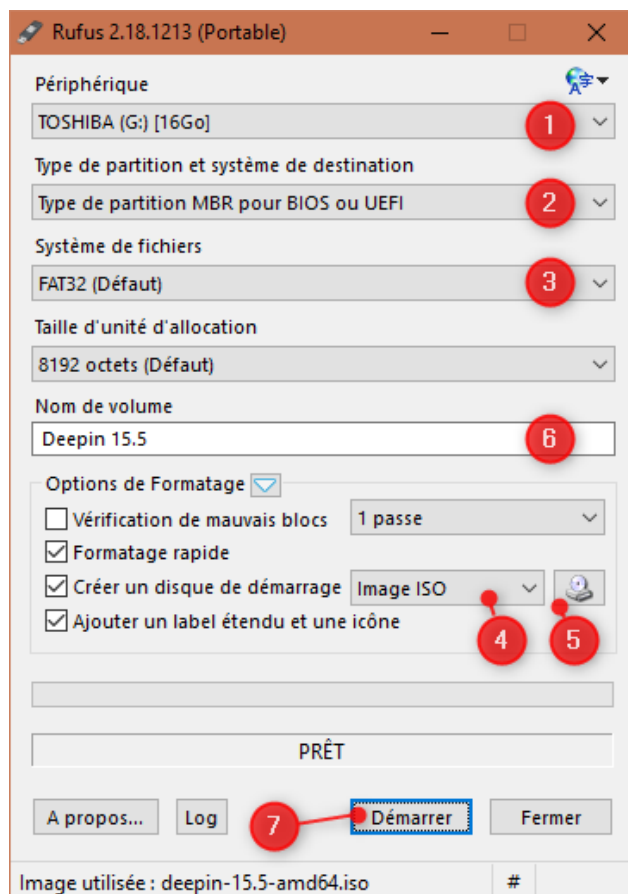
Par défaut, une image *iso* est un fichier destiné à être gravé sur un disque optique. Pour créer un media usb bootable, il ne suffit pas de copier le fichier iso sur la clé, malheureusement.

Pour créer un live usb (qui fera aussi office de media d'installation), il faut posséder une clé de minimum 4 Gio vierge, ou dont le contenu peut être perdu. En effet, le processus de création de la clé nécessite son formatage, détruisant de ce fait toutes les données qui y sont inscrites.

Pour nous aider à créer la clé usb, il faut télécharger l'outil *Rufus* <https://rufus.akeo.ie>. Prenez la version portable, elle ne nécessite pas d'installation et peut donc être mise à la corbeille après utilisation.

Dans la fenêtre Rufus:

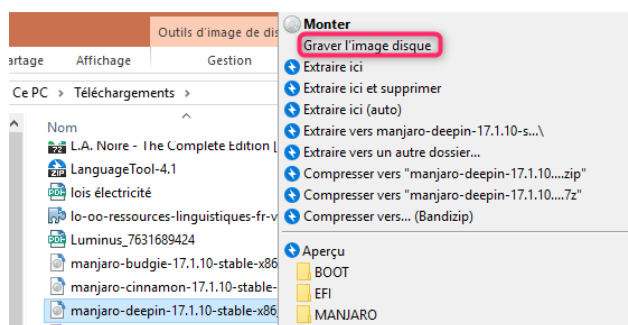
1. Sélectionnez la clé USB que vous souhaitez transformer. Elle se détecte automatiquement;
 2. Laissez le type de partitionnement =BIOS= (ancien système de démarrage des ordinateurs) et *EFI* (nouveau système);
 3. Sélectionnez le système de fichiers FAT32. Sinon, la clé risque de ne pas être reconnue au démarrage.
 4. Laissez les cases cochées. Dans le sélecteur de disque de démarrage (marqué *FreeDOS* par défaut), indiquez que vous souhaitez copier une image **iso**;
 5. Cliquez sur l'icône du cd pour choisir l'image iso du Linux à copier;
 6. Le nom de la clé prendra le nom de l'image iso sélectionnée.
1. Cliquez sur le bouton **Prêt** pour lancer la copie.



4.5.2.2. Créer un live CD

Pour créer un live cd (en pratique, un live dvd), il faut avoir un dvd vierge à graver et un graveur de disque (ce que tout ordinateur fait désormais). Ensuite, avec Windows, faites un clic droit sur l'image iso téléchargée et sélectionner l'option *graver l'image disque*.

Patiencez le temps que le graveur fasse son œuvre. Une fois la gravure terminée, vous pourrez utiliser votre disque dès le démarrage de l'ordinateur (ou de la machine virtuelle).



4.5.2.3. Utiliser un live CD/USB

Note

Vous pouvez utiliser une machine virtuelle pour tester Linux en Live CD. Démarrez simplement une machine virtuelle en plaçant le fichier iso dans le lecteur de disque virtuel (cf. plus haut). Malheureusement, ça ne marche pas avec les live USB.

L'utilisation d'un live usb (ou live cd) est très simple. *Avant* de démarrer l'ordinateur, insérez le media de démarrage. Ensuite allumez l'ordinateur et tapez plusieurs fois sur la touche **F12**. Cela va activer un sélecteur de démarrage, dans lequel vous allez choisir le périphérique. Le système Linux devrait démarrer.

En version Live CD, vous pourrez utiliser Linux directement, mais sans pouvoir installer des applications ou écrire des fichiers autres que ceux de votre disque dur. Cela en fait un environnement idéal pour tester votre distribution Linux! De plus, ce media permet aussi l'installation de Linux.

Pour terminer l'utilisation de Linux, éteignez l'ordinateur, puis retirez le media. Au prochain démarrage, vous allez retrouver l'ordinateur tel qu'il était!

4.6. L'installation en côte à côte

La troisième méthode est l'installation en côte à côte avec le système d'exploitation existant. (ce que l'on appelle le **dual boot**). Il s'agit ici d'installer **réellement** le système Linux sur le disque dur.

En contrepartie, l'espace du disque dur sera divisé (partitionné) pour permettre l'installation de Linux. Windows sera donc amputé de quelques Gio.

Mais le principal avantage de cette méthode est que le système sera complètement installé sur votre ordinateur, vous permettant d'apprécier au mieux les performances, tout en conservant l'accès à vos informations sous Windows (**attention**, Linux est capable d'accéder aux informations sur une partition Windows, mais pas l'inverse!).

Note

L'installation en dual boot va opérer un petit changement au démarrage de l'ordinateur: vous aurez le choix entre Linux et Windows au démarrage, pendant quelques secondes. Sachez que par défaut, c'est Linux qui démarrera en premier, mais il y a moyen de changer ce réglage.

4.7. L'installation en remplacement de Windows.





Enfin, la dernière méthode, c'est de sauter directement dans le bain! Il s'agit d'installer Linux sur l'ordinateur, en lieu et place de Windows. Coté avantage, vous aurez les pleines performances du système sur votre ordinateur. Cependant, l'installation du Linux *effacera totalement* les données présentes sur votre disque dur. Pensez à faire un backup de votre disque si vous n'êtes pas totalement sûr de vous, ou essayez une autre méthode avant celle-ci.

5. Utiliser la console

La console Linux est un outil indispensable dans la gestion de l'ordinateur. Le shell le plus connu et le plus utilisé (et installé par défaut) est le **BASH** (Bourne Again SHell), dérivé du Bourne Shell, le shell le plus ancien. La plupart des Linux utilisent BASH par défaut.

Le terme **console** (ou **terminal**) est employé pour définir l'ensemble des périphériques permettant à l'utilisateur d'interagir avec le système. Afin d'offrir plusieurs terminaux à l'utilisateur à partir du même ensemble physique, Linux fournit des *consoles virtuelles*.

Ces consoles virtuelles sont généralement de 6 ou 7, dont une console « graphique », utilisées par les environnements de bureau. Ceux-ci contiennent également des émulateurs de terminaux, appelés (à tort) *terminaux* eux-mêmes. Les terminaux sont accessibles via

les combinaisons de touche  +  +  à  (cette dernière étant la console graphique).

Il est aussi possible d'interagir à distance avec la ligne de commande Linux en utilisant des terminaux distants, avec **SSH**. Il est alors possible de commander un système Linux à partir d'une autre machine.

Une fois connecté à un terminal, un *shell* est lancé automatiquement. Il permet de saisir les différentes commandes textuelles que nous verrons plus tard.

5.1. Les types de shell

Sous Linux, nous pouvons utiliser différents types de shell, qui apportent tous des fonctionnalités plus ou moins similaires.

bash : Bourne-Again Shell, évolution du Bourne shell. Bash est de shell du projet **GNU**. C'est le shell que l'on voit le plus souvent dans la plupart des distributions Linux.

tcsh : Évolution de **csch**, lui-même évolution du Bourne Shell (**sh**), qui utilise une syntaxe d'écriture proche du C. Il est toutefois peu utilisé et on lui préfère Bash.

ksh : Korn Shell, développé par David Korn au laboratoire AT&T Bell. Il inclut un grand nombre des fonctionnalités du **csch** et est compatible Bourne Shell. C'est plus un langage informatique qu'un simple shell.

ash : Ou sa version spécifique à Debian, **dash**, est un interpréteur de commandes lui aussi dérivé du Bourne Shell, créé par Kenneth Almquist. Sa particularité est qu'il est extrêmement compact et rapide, ce qui en fait un shell de choix pour tout système Linux embarqué. Il ne semble toutefois plus maintenu.

zsh : Est un Bourne Shell étendu, avec beaucoup d'améliorations, reprenant la plupart des fonctions les plus pratiques de *bash*, *ksh* et *tcsh*. Il est de plus 100 % compatible avec *bash*. Il offre notamment l'autocomplétion des commandes, la correction automatique des commandes mal orthographiées, l'utilisation simplifiée de variables et de tableaux, la prise en charge de l'Unicode... **zsh** dispose aussi d'une foule de plugins et de thèmes, permettant ainsi de personnaliser son shell comme on le souhaite. Ce qui est d'ailleurs un inconvénient : l'installation de plusieurs plugins et autres thèmes auront tendance à complexifier et ralentir **zsh**. De plus, cet interpréteur de commandes nécessite la création d'un fichier de configuration relativement complexe pour être utilisé. Néanmoins, il existe le projet *oh-my-zsh* qui permet de simplifier la configuration de ce shell.

fish : Friendly Interactive SHell. La volonté de **fish** est claire et simple: disposer d'un shell interactif qui reprend **toutes** les meilleures fonctionnalités des autres shells! Néanmoins, il n'est pas compatible avec *Bash* et le scripting POSIX, car, selon les développeurs, la norme POSIX est contraignante et source de bugs. La syntaxe **fish** est plus propre et plus lisible.

Powershell : Dans sa version 6, Powershell est un nouveau venu dans la liste des shells Linux. Tout ce que nous avons vu au cours de Powershell sera compatible. Toutefois, Powershell pour Linux est encore jeune et très peu de Linuxiens l'utilisent.

Note

Personnellement, j'utilise **zsh** car il est plus intuitif à utiliser et permet une meilleure interaction avec le système d'exploitation. Toutefois, il est plus complexe à appréhender.

Pour les shells Linux, certains caractères ont un sens particulier.

Carac-tères	Signification
*	Correspond à 0 ou n caractères quelconques
?	Correspond à 1 caractère quelconque
[]	Correspond à une plage de caractères
~	Correspond au répertoire personnel

5.2. Les commandes indispensables

Les commandes Linux sont généralement des commandes courtes et dont leur nom est tiré d'une action. Elles ont des options qui modifient leur comportement. Ces options sont généralement non obligatoires et peuvent parfois recevoir des arguments.

5.2.1. Syntaxe d'une commande Linux

`commande` [-option] [argument1 | argument2]

Une commande représente un nom de fichier exécutable, qui est composé de 2 à 9 caractères et composé uniquement de lettres minuscules et/ou chiffres.

Une option :

- toute option doit être précédée du signe -;
- une option est constituée d'un seul caractère, minuscule ou majuscule;
- l'ordre des options n'a pas d'importance;
- les options sans paramètres peuvent être regroupées après un - unique;
- une option peut avoir un ou plusieurs paramètres;
- un paramètre d'option suit celle-ci séparée d'un espace;
- si une option possède plusieurs paramètres, ceux-ci doivent être séparés par une virgule ou regroupés entre guillemets et séparés par des espaces.

Un argument :

- représente souvent un nom de fichier ou un chemin d'accès à un fichier ou une entrée standard;

- l'ordre relatif des arguments peut affecter leur rôle ou leur importance selon des façons déterminées par la commande avec laquelle ils sont associés ;
- - précédé et suivi d'un espace ne doit être utilisé que pour désigner l'entrée standard ;
- . représente le répertoire courant ;
- .. représente le répertoire parent par rapport au répertoire de travail.

Chaque commande fournit un code de retour traduisant l'état de l'opération :


- **0** indique que la commande s'est exécutée sans erreur ;
- **≠0** indique une anomalie. Le manuel de la commande fournira les différents codes d'erreurs.

5.2.2. Les commandes de base

Note

Avec zsh, utilisez la touche  pour compléter votre commande ou les options (-). Vous serez surpris du résultat!

Commençons par une commande qui vous sera très utile :

man *cmd* : Cette commande affichera le manuel de la *cmd* passée en paramètre. `man man` affichera le manuel de la commande `man`. Pour quitter le manuel, appuyez sur .

[sudo] mandb : Cette commande est utilisée pour mettre à jour le manuel. Au démarrage du système, `mandb` est appelé automatiquement, mais il est parfois nécessaire de lancer la commande manuellement.

clear : Cette commande permet de nettoyer l'écran.

history : Cette commande permet de lister l'historique des commandes tapées précédemment. Chaque commande précédemment tapée est précédée d'un numéro. Par rappeler cette commande, il suffit de taper `history -n`, *n* étant le numéro de la commande. Pour rappeler la dernière commande passée, utilisez le raccourci `!!` qui est beaucoup plus rapide! Enfin, pour effacer l'historique, tapez `history -p`.

w : Affiche qui est connecté et fait quoi.

who : Affiche qui est connecté. Cette commande est surtout utile en environnement multi-utilisateur.

whoami : Affiche sous quel nom d'utilisateur vous êtes connecté au système.

su : **Super User**, permet de passer en *root* (notamment).

sudo *cmd* : Cette commande est utilisée fréquemment, car elle permet de passer une *cmd* en tant que *root*.

whatis *cmd* : Affiche la description courte du manuel de la commande.


uname : Affiche les informations système. Consultez le manuel pour connaître les différentes options.

echo *string* : Affiche un texte passé en paramètre à l'écran.

date : Affiche et règle la date courante.

cal : Affiche un calendrier, par défaut du mois en cours.

passwd : Cette commande permet à l'utilisateur de modifier son mot de passe. Les mots de passe sont stockés dans le fichier `/etc/shadow`. En ajoutant un nom d'utilisateur comme paramètre, cette commande permet de changer le mot de passe de l'utilisateur ciblé. **Attention**, seul un administrateur peut modifier le mot de passe des autres utilisateurs !

info : Encore une commande similaire à `man`, mais qui affiche les informations avec des liens entre les différentes pages du manuel. Cette commande étant relativement complexe à utiliser, tapez dans l'info sur . Un tutoriel va se lancer pour vous expliquer en détails le fonctionnement de cet outil. Une fois maîtrisé, `info` sera votre meilleur allié !

5.2.3. Les alias

Un alias est un moyen de mettre un nom simple sur une commande ou un ensemble de commandes. En tapant `alias` sans arguments, vous aurez une liste des alias courants. Pour créer un nouvel alias, on utilisera la syntaxe `alias [-s] nom="commande"`. Le `-s` permet d'enregistrer l'alias pour le réutiliser sans devoir le recréer. Pour supprimer un alias, on utilisera la commande `unalias nom`.

5.2.4. Commandes internes et externes

Sous Linux, le terminal peut lancer toutes sortes de commandes, aussi bien les commandes internes propres au shell que les commandes externes. Il est très compliqué de distinguer si la commande que l'on tape est une commande, un alias, un script... Une commande *interne* est une commande propre au shell, qui ne va pas déclencher un nouveau processus pour fonctionner, tandis qu'une commande *externe* va créer un processus enfant.

Toutefois, la commande `type cmd` permet d'en savoir plus.

Dans la même veine, la commande `whereis cmd` permet de localiser rapidement une commande externe.

5.3. Les flux de redirection et le piping

Par défaut, il existe trois flux de données. Le premier est le flux `in`, qui est le flux d'entrée au clavier et qui porte le n° de descripteur **0**. Le second est le flux `out`, qui est le flux de sortie standard sur la fenêtre de terminal et qui porte le n° de descripteur **1**. Enfin, le dernier flux est le flux d'erreurs `err`, qui porte le n° de descripteur **2**.

On peut toutefois rediriger le flux de données ailleurs, comme récupérer le résultat d'une commande dans un fichier.

En tapant `ls -a > result.txt`, on va envoyer le résultat de `ls -a` dans le fichier `result.txt`, en créant ce fichier dans le répertoire courant. Si ce fichier existe déjà, son contenu actuel sera supprimé et remplacé par le résultat de la commande.

En tapant `ls -a >> result.txt`, on va faire la même chose à une petite différence près : on va *ajouter* le résultat de la commande en fin de fichier, au lieu de le remplacer. C'est le mode **append**.

La redirection marche dans les deux sens : en tapant par exemple `sort < result.txt`, on envoie le contenu du fichier dans la commande `sort`. Le tri sera affiché à l'écran.

Bien entendu, on peut mixer les redirections. Si on veut avoir un listing trié des différents répertoires de l'ordinateur, on va taper la syntaxe suivante: `sort < rslt.txt > sorted.txt`.

La redirection `<<` va lire les caractères jusqu'à la rencontre d'une certaine chaîne de caractères. Par exemple: `cat << done [...]` je tape du texte jusqu'au mot clé indiqué (*done*).

5.3.1. Redirection des erreurs

Par défaut, les messages d'erreurs sont redirigés vers la sortie standard, mais on peut modifier ce comportement. Pour sauver les erreurs dans un rapport, on ajoutera `2> error.log` à nos commandes.

5.3.2. Piping

Le *Piping*, ou le chaînage de commandes, permet de lier la sortie d'une commande avec l'entrée d'une autre. Cette fonctionnalité décuple les possibilités de la console.

6. Organisation du système de fichiers

Le système de fichiers de Linux a une organisation totalement différente de celle de Windows. Sous Windows, nous avons l'habitude d'utiliser des *disques/*, qui peuvent être des disques matériels ou des partitions, contenant chacun un système de fichiers. Avec Linux, c'est différent. Le **FHS** (pour */Filesystem Hierarchy Standard*) est une convention qui organise l'arborescence des répertoires sous les systèmes Unix et Linux. Grâce à cette convention, lors de l'installation de programmes, les fichiers s'installent dans les bons répertoires, quelle que soit la distribution utilisée.

L'autre principe de base du FHS est que **tout est fichier**. Une imprimante, une partition de disque dur... seront considérés comme des fichiers dans lesquels on pourra écrire. Le pilote prendra en charge la conversion des informations en actions à effectuer.

/ (root) : Sous Linux, toute l'arborescence part de **/**, appelée *racine* ou *root*. On retrouvera sous la racine les principaux dossiers élémentaires du système.

/bin : Contient les programmes (*binaries*, les programmes exécutables) vitaux pour l'utilisateur.

/boot : Contient les fichiers servant au démarrage du système.

/dev : Contient tous les périphériques.

/etc : Contient les fichiers de configuration et des fichiers nécessaire au démarrage du système.

/home : Contient les répertoires personnels des utilisateurs.

/lib : Contient les bibliothèques (*libraries*) partagées.

/mnt : Est le répertoire de base pour les points de montage temporaires. Nous parlerons plus loin des points de montage.

/proc : Ce répertoire est un peu différent, car il constitue un point d'accès direct aux paramètres du noyau.

/root : C'est le répertoire personnel du super-utilisateur, nommé lui aussi *root*.

/sbin : Contient les programmes vitaux pour le super-utilisateur.

/tmp : Contient les fichiers et dossiers temporaires. Ce répertoire est vidé à chaque démarrage de l'ordinateur.

/usr : Le dossier *User System Resources* (souvent appelé à tort *user*) contient une série de dossiers qui ne sont pas vitaux pour le système.

/var : Contient des fichiers *variables*, qui changent fréquemment.

6.1. Les fichiers sous Linux

On entend par **fichier** la structure contenant des données utilisateur. Sous Linux, **tout est fichier**, même les disques durs et autres périphériques! Cette notion un peu particulière permet toutefois de simplifier énormément la gestion du système.

Les fichiers standards sont constitués d'une suite de caractères ou d'un flux d'octets dont le format n'est pas imposé par le système (comme Windows le fait), mais par les applications elles-mêmes.

Un répertoire est un fichier particulier, pouvant contenir d'autres fichiers (et donc de répertoires...). Il existe sept types de fichiers sous Linux, dont les trois plus importants sont :

- Les fichiers ordinaires ;
- Les répertoires ;
- les liens symboliques.

Les quatre autres types de fichiers sont manipulés principalement en programmation système, ils ne seront pas vus ici.

6.2. Les systèmes de fichiers

Un système de fichiers est une façon d'organiser et de stocker une arborescence sur un support physique. Chaque OS a développé son propre système de fichiers. Les principaux systèmes de fichiers sont :

FAT, FAT32, ExFAT : Pour *File Allocation Table*, ce système de fichiers a été conçu par Microsoft pour MS-DOS.

La Table d'Allocation des Fichiers est un index qui liste le contenu du disque, afin de stocker l'emplacement des fichiers sur celui-ci. Sur le disque dur, les fichiers sont stockés dans des *clusters*, des blocs d'une certaine taille comparable à des tiroirs. Étant donné que les blocs contenant les fichiers n'étant pas forcément contenu (phénomène de *fragmentation*), la FAT permet de conserver la structure des fichiers en créant des liens vers les différents blocs contenant les fichiers.

À l'origine, la FAT était un index de 16 bits acceptant des noms de fichiers de 8 caractères maximum, suivi d'une extension de 3 caractères.

Depuis Windows 95, la FAT a évolué vers un nouveau standard, la **FAT32**, qui fonctionne sur 32 bits. Cette évolution permet également d'écrire des noms de fichiers de maximum 255 caractères.

Grâce à sa conception, la FAT32 ne peut pas gérer des fichiers de plus de 2^{32} octets, soit 4 Go. Pour pallier ce problème, le format ExFAT a été créé.

Le format ExFAT est conçu à l'origine pour être utilisé sur les clés USB et autres supports de stockage externe. Étant un système de fichiers 64 bits, il permet le stockage de fichiers volumineux et le support de gros volumes. Toutefois, ce système de fichier prend plus de place qu'un système *NTFS* et c'est une technologie Microsoft soumise à des restrictions.

NTFS : La **New Technonolgy File System** de Microsoft a été développé pour la branche NT de Windows, et utilisé par défaut depuis Windows Xp.

Le NTFS étant basé sur un principe d'arbre binaire, la recherche d'un fichier est plus rapide que sur un système *FAT*, en octroyant la sensibilité à la casse ainsi que la compression native des fichiers sur le disque.

ext2, ext3, ext4 : Les systèmes de fichiers **ext** se base sur le principe de *l'extent*, une zone de stockage contigüe réservée pour un fichier. À chaque écriture du fichier, les données sont ajoutées à l'extent. Cela réduit (voire neutralise) la fragmentation des fichiers.

Chaque *extent* se caractérise par un **inode**, ou nœud d'index, une structure de données contenant les informations à propos d'un fichier. Chaque fichier possède son *inode*

unique, contenant des informations spécifique sur le propriétaire, les droits d'accès... Les formats ext2 et ext3 sont des systèmes de fichiers obsolètes. La plupart des systèmes Linux utilisent **ext4**.

La version 4 de ext est, selon les développeurs eux-mêmes, une solution transitoire vers un nouveau format de fichiers extrêmement performant, mais encore en tests, le **btrfs**. Ext4 gère des volumes de données allant jusqu'à 2^{60} octets et permet la préallocation contigüe des extents pour les gros fichiers, limitant ainsi la fragmentation.

ReiserFS : Le système **ReiserFS** est un système de fichiers moins utilisé que l' *ext4* mais il est plus performant sur le traitement de répertoires contenant des milliers de fichiers de petite taille. ReiserFS permet aussi l'agrandissement à chaud et la diminution à froid de la taille des partitions. Cela signifie que l'on peut ajouter un disque dur **pendant que le système est en fonctionnement** et l'intégrer à l'arborescence.

BTRFS : Le **btrfs**, prononcé *ButterFS* est un système de fichiers récent, non terminé à ce jour, mais déjà fonctionnel.

Btrfs offre des fonctionnalités intéressantes, absentes sur les autres systèmes de fichiers, à savoir la création de *snapshots* (qui garantit de pouvoir faire une sauvegarde des données même si on continue de travailler sur ces données!) et la gestion des sommes de contrôles (pour vérifier l'intégrité des données en temps réel).

Btrfs se base sur les extents et gère une notion de « sous-volumes » permettant plus d'arborescences simultanément, indépendantes du système principal. Les sous-volumes permettent aussi la limitation de l'espace disque par *quotas*.

Btrfs permet également, comme ReiserFS, le redimensionnement des partitions à chaud.

6.3. Les chemins

Le système de fichiers de Linux est une représentation virtuelle de *tout* le système, autant logiciel que matériel. Tous les répertoires sont reliés à la racine et l'utilisateur peut naviguer au sein de cette hiérarchie pour accéder à toutes les ressources. À chaque instant, l'utilisateur se situe quelque part dans cette hiérarchie. La commande `pwd` permet de connaître l'endroit de cette hiérarchie où se situe l'utilisateur à ce moment.

Pour se déplacer dans cette hiérarchie, il faut connaître les trois modes de déplacements possibles: par le chemin *absolu* (toujours par rapport à la racine), par les chemins *relatifs* (par rapport à l'endroit où l'on se situe actuellement), ou par les chemins *personnels* (par rapport aux répertoires utilisateurs).

6.3.1. Chemin absolu

Le chemin **absolu** d'un dossier ou d'un fichier est donc son emplacement *par rapport à la racine du système*. La séparation entre les dossiers se fait par des `/`, contrairement à Windows qui utilise les `\`.

Note

Un chemin absolu est facilement identifiable, car il commencera **toujours** par la racine `/`.

6.3.2. Chemin relatif

Le chemin **relatif** est le parcours à effectuer dans la hiérarchie *depuis l'endroit où l'on se trouve*. On utilisera la notation `..` pour remonter d'un niveau dans l'arborescence et la notation `.` pour indiquer le répertoire actuel.

6.3.3. Chemin personnel

Cette troisième façon de définir le chemin se réfère au **répertoire personnel de l'utilisateur**. On emploie cette technique principalement pour accéder aux données dans les sous-répertoires de `/home`. Un chemin personnel commence par le `~` suivi du nom de l'utilisateur. Si le nom n'est pas spécifié, l'identité de l'utilisateur connecté sera utilisée.

6.4. Les points de montage

Nous avons vu précédemment l'arborescence d'un système Linux. En réalité, chaque partition ou disque amovible abrite un système de fichiers « physique ». Pour cela, une partition est désignée comme étant la partition racine (**root**) et va accueillir les autres systèmes de fichiers. Pour accéder au contenu d'un système de fichiers physique, il faut l'attacher à l'arborescence principale. Cette opération s'appelle le **montage**. Elle s'effectue au moyen de la commande `mount`.

Avec une interface graphique, la plupart des périphériques amovibles sont montés et démontés automatiquement. Mais nous pouvons également monter des disques réseaux. Cette commande peut être très pratique pour accéder aux répertoires partagés de Virtualbox.

`mount` : Cette commande permet de monter un (nommé `sdx`, où `x` est une lettre) sur un existant. Par exemple, pour monter le dossier partagé *Documents* depuis Windows dans une machine virtuelle Linux, il faut taper :

```
1 mount -t vboxsf Documents ~/Public
```

Zsh

Détaillons cette commande: le `-t vboxsf` indique le type de système de fichiers. Il s'agit ici d'un type spécial propre à Virtualbox, d'où l'importance de le spécifier. Dans la plupart des cas, on n'a pas besoin de spécifier le type de système de fichiers. Ensuite, le *Documents* représente le nom du répertoire partagé par Virtualbox. Enfin, le `/Public` représente le chemin vers le répertoire Public. Le point de montage doit être nécessairement vide!

`umount` : Cette commande permet de démonter le système de fichiers.

6.5. Les commandes relatives aux fichiers

ls : Affiche la liste des fichiers d'un répertoire. Avec l'option `-l`, on affiche une liste détaillée. Notez la différence de couleurs pour les fichiers et les répertoires. Nous verrons plus loin leur signification. En tapant `ls -l`, vous verrez d'abord la taille totale des fichiers, ainsi qu'une série de lettres ou de traits représentant les permissions.

- La première lettre indiquera le type de fichier (*d/ pour /directory*);
- Les trois groupes de trois lettres (`rwX`) représentent les permissions du fichier (nous les verrons également plus loin);
- Le nombre de liens physiques;
- Le nom du propriétaire du fichier;
- Le nom du groupe propriétaire;
- La taille en octets;
- La date de dernière modification.

Passez les paramètres :

- `-a` pour afficher les fichiers cachés ;
- `-l` pour afficher la liste détaillée ;
- `-h` pour afficher les tailles en version « human readable » ;
- `-R` pour afficher les sous-dossiers (récursivité).

cd : Change Directory. Cette commande permet de naviguer dans l'arborescence en changeant de dossier. `cd` accepte les chemins absolus et les chemins relatifs. Quelques raccourcis utiles avec `cd` :

- `=cd ~` mène au répertoire utilisateur (le « home ») ;
- `=cd ..` remonte au répertoire parent ;
- `=cd -` retourne sur le chemin précédent.

Note

Certains shells (comme FISH ou ZSH) acceptent le **globbing**, c.-à-d. l'utilisation de caractères joker. Par exemple, `cd ../../` remonte de deux répertoires.

cat **et** **tac** : La commande `cat` fichier affiche le contenu du fichier. La commande `tac` fichier affiche le contenu du fichier, mais **lu de bas en haut**. Cette dernière commande est utile pour afficher les premières lignes d'un fichier. La commande `cat` permet également de concaténer plusieurs fichiers.

file : Puisque la notion d'*extension* n'existe pas sous Linux, il peut être hasardeux de se fier uniquement au nom d'un fichier pour déterminer son contenu. La commande `file` fichier permet d'afficher cette information.

du : **Disk Usage**, qui précise l'espace disque que prend chaque fichier ou dossier.

df : **Disk Free**, qui précise l'espace disque disponible sur le système de fichiers dont l'utilisateur possède l'accès en lecture.

pwd : **Print Working Directory**. Cette commande affiche le chemin absolu du répertoire actuel.

head : Affiche le début d'un fichier. Combiné avec l'option `-n nombre`, permet d'afficher le nombre de lignes spécifié.

tail : Similaire à `head`, mais affiche la fin du fichier. Avec l'option `-f`, permet d'activer le *following*, l'affichage en temps réel de la fin du fichier, ce qui est fort pratique pour suivre l'évolution des fichiers logs.

touch fichier : Crée un fichier vide. Le but premier de `touch` est de modifier les dates d'accès et de modification du fichier s'il existe déjà.

mkdir dossier : **Make Directory**. Crée un dossier. Les noms de dossiers peuvent être enchaînés. Ex: `mkdir ~/toto/{tutu,tata,titi}` va créer le dossier *toto* ainsi que les sous-dossiers indiqués.

cp src dst : Copie un fichier. L'option `-R` permet de copier un dossier et tous ses sous-dossiers.

mv src dst : Déplace un fichier (ou un dossier avec l'option `-R`). Cette commande permet aussi de renommer un fichier ou un dossier.

rm fichier : Supprime des fichiers. Avec l'option **-f**, on force la suppression. Avec l'option **-i**, on demande une confirmation avant suppression. L'option **-r** permet la suppression des dossiers.

rmdir : **Remove Directory**. Supprime un dossier **uniquement** s'il est vide.

wc fichier : **Word Count**. Cette commande mérite qu'on s'y attarde un peu. Elle permet de compter le nombre de lignes (**-l**), de mots (**-w**) ou de caractères (**-m**). L'option **-c** affiche la taille en bits du fichier. On peut aussi facilement compter le nombre de fichiers ou de dossiers dans un répertoire en « pipant » la sortie d'une commande dans `wc~: ~ls`
| `wc`.

more fichier **et** **less fichier** : Ces deux commandes permettent d'afficher le contenu d'un fichier texte en l'affichant page par page. **less** est une fonction plus évoluée que **more**.

7. Les droits d'accès aux fichiers

7.1. Concepts de compte utilisateur et de groupe

Le système Linux est nativement multi-utilisateur. Chaque personne doit posséder un compte qui les identifie clairement sur le système. Chaque utilisateur doit être membre d'un groupe (il peut être seul dans son groupe) et peut appartenir à plusieurs groupes différents. Ces groupes permettent à plusieurs utilisateurs appartenant à un même groupe de partager des fichiers. Les utilisateurs ne sont pas tous égaux dans le monde Unix. Il existe trois types de comptes :

root (UID 0) : C'est l'utilisateur administratif le plus important. C'est l'équivalent de *Dieu* pour le système. Il n'est pas concerné par les droits d'accès aux fichiers et peut faire à peu près tout. Ce compte utilisateur est utile pour les tâches d'administration du système et se devrait d'être au mieux désactivé, ou au moins protégé par un mot de passe solide.

Les comptes gestionnaires (UID 1 à 999) : Cette série de comptes ne représentent pas des personnes, mais des logiciels ou des services (appelés *daemons*), afin de faciliter l'accès aux ressources dudit programme.

Les comptes utilisateurs (UID 1000+) : Tous les autres comptes utilisateur sont associés à des personnes physiques. Ces utilisateurs n'ont pas le droit de modifier le système, sauf s'ils appartiennent à un groupe spécifique leur donnant ce droit.

De même que les comptes utilisateurs, il existe différents types de groupes permettant de donner des droits aux utilisateurs membres.

7.2. Les droits sous Linux

Les permissions d'accès aux fichiers déterminent les actions que peuvent entreprendre les utilisateurs. Les droits d'accès sont définis en trois groupes :

- Les droits du propriétaire du fichier, c.-à-d. l'utilisateur qui l'a créé ;
- Les droits du groupe principal du propriétaire, qui peut être modifié par le propriétaire pour partager les droits avec les membres du groupe choisi ;
- Les droits des autres utilisateurs.

Il y a trois types de droits :

- Le droit en *lecture* (**r**). Pour un dossier, permet de lister le contenu du dossier ;
- Le droit en *écriture* (**w**). Pour un dossier, permet de modifier le contenu du dossier ;
- Le droit en *exécution* (**x**). Pour un dossier, permet d'accéder aux sous-dossiers et aux fichiers.

En tapant la commande `ls -l`, on voit sur la gauche les différents droits répartis par groupes.

```
1 +rwxr--r-- 1 cedric prof 04-05-22 10:22 toto
```

Zsh

Le premier caractère ou trait représente le type de fichier. Le premier groupe de 3 caractères (rwx) représente les droits du propriétaire (**u**). Les 3 suivants (rw-) représentent les droits du groupe (**g**) et les 3 derniers (- - -) les droits des autres (**o**).

7.3. Gestion des droits

`id user` : Cette commande permet de connaître l'UID de l'utilisateur ciblé, son GID principal ainsi que les GID secondaires.

groups : Cette commande permet de lister tous les groupes existants, ou, avec les bons paramètres, la liste des groupes de l'utilisateur ciblé.

chgrp : Par défaut, chaque fichier appartient au groupe principal de l'utilisateur propriétaire du fichier, à moins qu'un droit spécial ne soit positionné (SGID). La commande **chgrp** permet de modifier ce groupe. Le propriétaire peut « céder » ce fichier à n'importe quel groupe auquel il appartient.

```
1 chgrp [-R] <groupe> <fichier>
```

 Zsh

L'option **-R** indique à la commande d'appliquer la modification de façon récursive.

useradd, userdel, usermod : Pour ajouter un compte utilisateur, utilisez la commande **useradd** suivi du nom de l'utilisateur en minuscule. Selon les différentes options passées, il est tout à fait possible de créer un compte administrateur (en ajoutant **-G wheel**, *wheel* étant le groupe conférant les droits superutilisateur), de créer les sous-répertoires utiles à l'utilisateur (**-m**)...

Pour modifier un utilisateur existant, on utilisera la commande **usermod** avec les mêmes options. Pour supprimer un utilisateur, on utilisera la commande **userdel**.

Note

Sous Debian et ses dérivés, un script nommé **adduser** permet de faire les choses de manière interactive. Mais ce script n'est pas disponible sur les autres plateformes.

Note

Par mesure de précaution, on utilisera la commande **passwd** pour assigner un mot de passe à l'utilisateur. On pourrait le faire dans la commande **useradd**, mais cela pose deux problèmes: Le premier est que le mot de passe apparaît en clair dans la commande, le second est qu'il faut manuellement encrypter le mot de passe!

passwd : Cette commande permet de modifier le mot de passe d'un utilisateur, de désactiver l'utilisateur (avec l'option **-l**), ou encore de forcer l'utilisateur à modifier son mot de passe.

groupadd, groupdel, groupmod : De manière similaire à **useradd** et consort, ces commandes permettent de créer, supprimer ou modifier des groupes.

chmod : Cette commande permet de changer les droits (ou *modes*) des fichiers. Seul le propriétaire (ou le superutilisateur) a le droit de modifier les modes du fichier.

```
1 chmod [-R] <droits> <fichier...>
```

 Zsh

Les droits se notent de deux façons différentes : soit en notation symbolique, plus longue, mais plus compréhensible, soit en notation octale, plus courte, mais plus dangereuse si on ne sait pas ce que l'on fait.

- La **notation symbolique** utilise les caractères **r**, **w** et **x** pour désigner les droits, ainsi que **u** pour l'utilisateur propriétaire, **g** pour le groupe et **o** pour les autres. Elle utilise également le **a** pour regrouper la notation *ugo*.

Pour ajouter un droit à celui existant, nous utiliserons le symbole **+**. De façon analogue, le symbole **-** supprimera les droits et le symbole **=** définit les nouveaux droits en supprimant les anciens.

```
1 chmod u+x toto #ajoutera le droit d'exécution au propriétaire
```

 Zsh

- La **notation octale** est un codage sur 3 bits des droits de chaque entité. **x** possède un poids binaire de 1, **w** un poids binaire de 2 et **r** un poids binaire de 4. La combinaison *rwX* de chaque entité correspond donc à un chiffre en octal.

```
1 chmod 754 toto #le propriétaire a rwx, le groupe r-x et les autres --x
```

 Zsh

chown : D'une façon similaire à **chgrp**, cette commande permet de modifier le propriétaire d'un fichier. Seul un superutilisateur ou le propriétaire du fichier peut utiliser cette commande.

umask : Cette commande permet d'initialiser les droits pour chaque nouveau document créé. Le masque passé en paramètre de la commande va *supprimer* aux droits existants les modes qui y sont inscrits.

8. Gestion des processus

Un *processus* (ou une *tâche*) est un programme ou une commande en cours d'exécution sur un système. Il existe deux types de processus : les tâches actives et les *démons*. Un démon est un processus en arrière-plan qui s'exécute en permanence sur le système. Souvent les démons ont un nom terminant par *d*, comme **cupsd** qui est le démon de gestion des impressions. Chaque processus est identifié par un numéro unique, le *PID* (Process IDentifier). Ce n° est fourni par le noyau Linux. Chaque processus est lancé par un processus parent, sauf le processus *init* ou *systemd* qui possède le PID **1**, car il est le premier processus lancé au démarrage du système.

ps : La commande `ps` permet de voir la liste des processus du système. Par défaut, cette commande affiche uniquement les processus lancés par l'utilisateur à partir du terminal. Si vous regardez dans le manuel, vous trouverez tellement d'options et de paramètres qu'il est quasi impossible de tout retenir. Toutefois, on utilise souvent les commandes `ps aux` et `ps -ef`.

top : La commande `top` permet de visualiser l'activité du processeur en temps réel. C'est en quelque sorte une amélioration de la commande `ps` plus interactive et plus compréhensible.

kill : La commande `kill`, ainsi que `killall` permettent d'envoyer des *signaux* aux processus. Par défaut, ces commandes envoient un signal **15**, qui est un signal ordonnant au processus de se terminer. La commande `kill` demande un PID, tandis que `killall` demande une commande. La liste des signaux peut être trouvée dans la section 7 du manuel pour signal (en clair : `man 7 signal`).

Note

On peut « tuer » des processus avec `top` en appuyant sur la touche k

9. Outils divers

9.1. find et locate

La commande `locate` permet de rechercher dans la base des inodes un fichier et en indique le chemin. Cette commande est très rapide, mais ne renvoie que des résultats appartenant à la base de données interne du système. Cette base de données peut ne pas être à jour et donc ne pas renvoyer le résultat escompté. La commande `updatedb` permet de forcer la mise à jour de la base de données des inodes.

La commande `find` permet une recherche plus en profondeur et permet également d'effectuer certaines actions. Néanmoins, la syntaxe de la commande `find` ne suit pas les règles de syntaxe des commandes Linux !

```
1 find <chemin> <critères> action
```

 Zsh

9.1.1. Chemin de recherche de find

On spécifie à `find` un ou plusieurs chemins de recherche. `find` se charge de parcourir toute l'arborescence dans les répertoires spécifiés. Si aucun chemin n'est spécifié, `find` recherche dans le répertoire courant.

9.1.2. Expressions de sélection de find

Les critères de sélection sont introduits avec un tiret suivi de leur paramètre.

`-name <motif>` : Le critère `-name` indique à `find` de chercher des noms de fichiers. Le motif peut contenir des caractères génériques. Pour éviter les erreurs d'interprétation, on convient d'englober le motif par des guillemets ().

```
1 find _usr -name `s*` 2>/dev_null #les erreurs sont envoyées dans le vide
```

 Zsh

`-size [+<taille>[bck]]` : Le critère `-size` permet de rechercher des fichiers selon leur taille en *blocs*. Par défaut, un bloc vaut 512 octets, mais on peut définir la taille des blocs en notant l'unité. `find _usr -size +12k 2>/dev_null #` recherche les fichiers de + de 12k. Les symboles **+** et **-** qui précèdent la taille indique s'il faut rechercher des fichiers de **plus** de x blocs, ou de **moins** de x blocs.

`-mtime [+<jours>]` : Le critère `-mtime` spécifie la date de dernière modification du fichier.

`-perm <droits>` : Ce critère permet de retrouver les fichiers avec des droits particuliers.

`-newer <fichier>` : Ce critère permet de rechercher des fichiers ayant une date de modification plus récente que le fichier passé en paramètre.

`-user <utilisateur>` : Permet de rechercher les fichiers appartenant à l'utilisateur indiqué.

`-group <groupe>` : Permet de rechercher les fichiers appartenant au groupe indiqué.

Les opérateurs booléens : `-and` ou `-a` effectue un *et* logique. `-or` ou `-o` effectue un *ou* logique. `-not` ou `!` effectue une inversion.

9.1.3. Actions de find

`-print` : C'est l'action par défaut qui va afficher à l'écran toutes les occurrences trouvées.

`-exec <commande {}>` : Pour chaque occurrence trouvée, l'option `-exec` va exécuter la commande passée en paramètre. Les accolades sont utilisées pour spécifier le chemin

du fichier trouvé par `find`. Pour délimiter la fin de la directive `-exec`, on termine **obligatoirement** la commande par la séquence de caractère `\`.

`-ok <commande {} \>` : Similaire à `-exec` à la seule différence que le système demandera une confirmation avant d'effectuer la commande passée en paramètre. **Attention**, les demandes de confirmations sont envoyées sur le flux d'erreur, il ne faut donc pas le rediriger!

`-ls` : Cette action affiche les informations détaillées des fichiers trouvés.

9.2. `grep`, `cut`, `sort`

La commande `grep` permet de rechercher des informations à l'intérieur d'un fichier ou d'un *pipe* selon un filtre. `grep options filtre fichier`.

La commande `cut` permet d'extraire des colonnes ou des champs sélectionnés.

La commande `sort` permet de trier les lignes passées en entrée et retourne un résultat trié.

10. Notions de shell scripting avec ZSH

Nous l'avons vu précédemment, un *shell* est un interpréteur de commandes, qui va boucler indéfiniment sur les actions suivantes :

- la lecture d'une ligne entrée par l'utilisateur ;
- l'interprétation de cette ligne comme une demande d'exécution d'un programme avec des paramètres ;
- le lancement du programme avec le passage des paramètres.

Le shell est bien plus qu'un interpréteur de commandes. Il s'agit d'un véritable environnement de programmation, permettant de définir des variables, des fonctions, des instructions complexes et même des programmes complets !

Quand on exécute un fichier de script (en l'invoquant par son nom), le shell interactif va lancer l'exécution d'un shell identique à lui-même pour exécuter les commandes contenues dans le fichier. On peut toutefois forcer l'exécution d'un shell particulier en utilisant un *shebang* en début de fichier :

```
1 #! /bin/bash # pour utiliser un script bash
2 #! /bin/sh   # pour utiliser un script shell
3 #! /usr/bin/python # pour utiliser python
```

 Zsh

Note

fish est un shell différent des autres, qui possède sa propre syntaxe d'écriture. Les scripts *fish* sont donc incompatibles avec les autres shells. C'est pourquoi nous verrons ici le scripting avec *zsh*, un shell tout aussi performant que *fish* et qui possède des améliorations notables par rapport à *bash*. Toutefois, nous ne verrons pas toutes les possibilités de *zsh*.

10.1. Écrire notre premier script

Pour réaliser un script shell, nous allons créer un simple fichier texte auquel on donnera communément l'extension *.sh* dans lequel on entre les commandes à exécuter.

```
1 #!/bin/zsh -f
2 # Hello.sh
3
4 # First script
5 # By VCO
6
7 echo "hello"
8 echo "world"
```

 Zsh

Note

Pour écrire plusieurs instructions sur la même ligne, il faut les séparer par un point-virgule (;).

N'oublions pas de rendre ce script exécutable en changeant les droits : `chmod a+x hello.sh`. Comme exercice, tentez de prédire chacune des lignes du script suivant :

```

1  #!/bin/zsh
2
3  # hello2.sh
4  # By VCO
5  # Que font ces différentes lignes ?
6
7  echo "Hello      World"
8  echo "Hello World"
9  echo "Hello * World"
10 echo Hello * World
11 echo Hello      World
12 echo "Hello" World
13 echo Hello "    " World
14 echo "Hello "*" World"
15 echo `hello` world
16 echo 'hello' world

```

10.2. Les variables

Le shell permet de manipuler des variables. Les variables n'ont pas besoin d'être déclarées et elles ne sont pas typées. Pour affecter une valeur à une variable, il suffit de taper le nom de la variable, suivi du signe `=` et de la valeur que l'on désire affecter à la variable **le tout sans espaces !**

Pour référencer la valeur d'une variable, on utilise la notation avec le symbole `$` suivi du nom de la variable. Par exemple :

```

1  MYVAR="Hello" # ⚠ pas d'espace autour du signe = !!
2  echo $MYVAR

```

Toutes les variables sont locales à un shell ; elles ne peuvent donc pas être modifiées par un sous-shell. Les variables exportées (avec la commande `export`) sont instanciées dans tous les sous-shells avec leur valeur. Les variables exportées via un script seront instanciées dans le shell qui les exporte.

10.3. La commande `read`

Nous pouvons affecter une variable interactivement en utilisant la commande `read` qui va demander une entrée à l'utilisateur :

```

1  #!/usr/bin/zsh -f
2  # by VCO
3  echo -n "Quel est votre nom ?: "
4  read NAME
5  echo "Bonjour $NAME ! Comment vas-tu aujourd'hui ?"

```

Analysons maintenant le code suivant :

```

1 #!/usr/bin/zsh -f
2 #by VCO
3 echo -n "Quel est votre nom?: "
4 read NAME
5 echo "Bonjour $NAME !"
6 echo "Ce script va créer un fichier nommé $NAME_fic"
7 touch "$NAME_fic"

```

Pour résoudre ce problème, il suffit d'entourer la variable d'accolades.

```

1 #!/usr/bin/zsh -f
2 #by VCO
3 echo -n "Quel est votre nom?: "
4 read NAME
5 echo "Bonjour $NAME !"
6 echo "Ce script va créer un fichier nommé ${NAME}_fic"
7 touch "${NAME}_fic"

```

10.4. Les commentaires et les « quotes »

Un commentaire sera précédé du symbole `#`. Il n'existe pas de commentaires multi lignes.

Pour écrire du texte dans un script, il faut l'entourer de *quotes*. Il existe plusieurs types de quotes :

- Les *simple quotes* `'` protègent le texte qu'elles entourent de toute transformation de la part du shell. Elles banalisent tous les caractères.

```

1 n=1
2 echo '$n' # affichera $n

```

- Les *double quotes* `"` permettent au shell d'interpréter leur contenu. Elles banalisent le contenu de tous les caractères sauf les `\`, les `$` et les `'`.

```

1 n=1
2 echo "$n" # affichera 1

```

Le `\` permet d'enlever toute signification au symbole qu'il précède.

- Les *back quotes* ``` permettent de faire des substitutions de commande. La substitution consiste à écrire une commande entourée du signe ```. Sur rencontre d'une commande *back quotée*, le shell exécute cette commande et remplace le texte de la commande par son résultat.

```

1 echo `pwd`
2 # affichera le résultat de pwd

```

Note

Avec **zsh**, les back quotes peuvent être remplacées par `$()`.

10.5. Passage de paramètres

Il est possible de passer des paramètres à un script. Les neuf premiers paramètres sont des variables portant les noms 1 à 9. Pour les référencer, on utilise la syntaxe des variables habituelles. Par exemple le code suivant :

```
1 #!/bin/zsh -f
2 # parameters
3 #by VCO
4
5 echo 'Ce script va copier le fichier passé en paramètre dans le répertoire BACKUP'
6 mkdir -pv ~/BACKUP
7 cp -i $1 ~/BACKUP
```

Nous pourrions l'invoquer par `backup toto.txt`

Il n'y a pas moyen de référencer directement les variables au-delà du neuvième. On verra plus loin comment procéder avec la commande `shift`. Par contre, pour référencer tous les arguments, on peut utiliser la variable spéciale **\$***

```
1 #!/bin/zsh
2 echo $*
```

Enfin il faut savoir que la variable **\$0** contiendra toujours le nom du script.

Il existe d'autres variables spéciales :

\$# : est le nombre de paramètres qui sont passés dans le script ;

\$@ : permet de récupérer **tous les paramètres** en une seule fois ;

\$* : est une version similaire à **\$@**, mais qui ne préserve pas les espaces ni les *quotes*. Tous les paramètres passés avec des espaces seront « éclatés ». En règle générale, il est préférable d'utiliser **\$@** ;

```
1 \#!/usr/bin_zsh -f
2 \#by VCO
3
4 clear
5 echo "Ce script est appelé avec \${#} paramètres"
6 echo "Le script se nomme $0"
7 echo "Les deux premiers paramètres sont: $1 et $2"
8 echo "Tous les paramètres sont $@"
```

\$? : permet d'avoir le code de retour de la dernière commande exécutée.

\$\$ et **#!** : permettent de connaître les PID. **\$\$** contiendra le PID du shell actif, tandis que **#!** contiendra le PID du dernier *job* exécuté en arrière-plan.

10.6. La commande shift

La commande `shift` a pour effet de décaler tous les paramètres **vers la gauche**. De ce fait, **\$2** devient **\$1**, **\$3** devient **\$2**... et le dixième paramètre, jusque-là inaccessible devient **\$9**. De plus, **\$#** diminue de 1.

10.7. Les tableaux

Le shell permet aussi de gérer des tableaux. Les tableaux créés sont des tableaux dynamiques indicés à partir de 0 (ou 1 avec `zsh`). La création d'un tableau se fait par :

```
1 tableau[n]=valeur \# affectation
2 echo ${tableau[n]} \# lecture de l'élément n
3 echo ${tableau[*]} \# lecture de tous les éléments
4 echo ${#tableau[*]} \# affiche le nombre d'éléments du tableau
```

10.8. Les structures de contrôles

Sous Linux, chaque programme peut, après son exécution, transmettre une valeur au processus appelant. En dernière instruction, un programme exécute `exit (n)`, où *n* est la valeur à transmettre. L'appelant exécute quant à lui la commande `wait(&status)`, où *status* est une variable dans laquelle le noyau place la valeur transmise.

Ces valeurs ainsi transmises servent de code de retour des programmes : elles servent à donner une indication sur la façon dont s'est déroulée l'exécution du programme. Par convention, la valeur **0** signifie que l'exécution du programme s'est bien déroulée. Une valeur différente de 0 signifie qu'il y a eu une erreur, la valeur étant un code choisi par le développeur de l'application.

Le shell récupère également ces codes de retour, et les place dans la variable spéciale **\$?**, que l'on peut manipuler comme n'importe quelle variable.

10.8.1. Les structures **&&** et **||**

La structure **&&** permet de créer un pipeline géré par les codes de retour. Si le premier pipeline rend un code de retour de 0, alors le second sera exécuté.

La structure **||** permet de créer un pipeline conditionnel, inverse au premier. Si le premier pipeline rend un code de retour différent de 0, alors le second sera exécuté.

10.8.2. La commande `test`

Cette commande très importante n'est pourtant pas utilisée dans sa forme primaire, mais avec son alias **[**. **[** est un lien symbolique vers `test`, communément employé pour rendre les scripts plus lisibles.

Pour écrire un test, il faut respecter la syntaxe suivante, **en tenant compte des espaces**.

```
1 [ $foo="bar" ]
```

10.8.2.1. Prédicats

Les prédicats sont des assertions de test. Selon que l'on va tester du texte ou des nombres, les prédicats vont s'écrire différemment. Si on travaille sur des chaînes de caractères, nous allons écrire :

<code>s1=s2</code>	TRUE si s1 est égal à s2
<code>s1 != s2</code>	TRUE si s1 est différent de s2
<code>-n s1</code>	TRUE si s1 est non vide
<code>-z s1</code>	TRUE si s1 est vide

Si on travaille sur des nombres, nous allons écrire :

n1 -eq n2	TRUE si n1 est égal à n2
n1 -ne n2	TRUE si n1 est différent de n2
n1 -gt n2	TRUE si n1 est plus grand que n2
n1 -ge n2	TRUE si n1 est plus grand ou égal à n2
n1 -lt n2	TRUE si n1 est plus petit que n2
n1 -le n2	TRUE si n1 est plus petit ou égal à n2

Si on travaille sur des fichiers :

-r fichier	TRUE si le fichier existe et qu'on ait le droit en lecture
-w fichier	TRUE si le fichier existe et qu'on ait de droit en écriture
-f fichier	TRUE si le fichier existe et que c'est un fichier
-d fichier	TRUE si le fichier existe et que c'est un répertoire
-s fichier	TRUE si le fichier existe et que sa taille est non nulle

Enfin, pour combiner les prédicats :

!	not
-a	and
-o	or
()	parenthèses

10.8.3. La commande `expr`

La commande `expr` permet de réaliser des calculs arithmétiques, des comparaisons de nombres entiers et de la reconnaissance de *patterns*. Cette commande interprète ses arguments comme étant une expression et imprime le résultat sur la sortie standard. Chaque opérande ou opérateur de la liste d'arguments doit être passé comme un seul paramètre (autrement dit, **avec des espaces**).

```
1 expr 2 + 3
```

 Zsh

10.8.4. La commande `exit`

La commande `exit` permet de terminer un shell en transmettant un code de retour. Si `exit` est invoqué avec un paramètre, c'est ce paramètre qui est pris comme code de retour, sinon ce sera le code de retour de la dernière commande exécutée.

10.9. La structure `if-elif-else-fi`

La syntaxe de la structure `if` est un peu particulière, comme toutes les syntaxes des différentes structures. Cette structure se termine par **fi** (*if* à l'envers) et chaque *branche* de la structure se comporte comme une nouvelle instruction. Cela signifie que chaque élément doit commencer sur une nouvelle ligne, où il faut séparer chaque élément par un `;`.

```
1 if [ test ] # attention aux espaces!!
2 then
3     # if-code
4 elif [ test ]
5     # elif-code
6 else
7     # else-code
8 fi
```

 Zsh

10.10. La structure case-esac

La structure `case` est une structure *if-elif-elif-...-else-fi* comportant plusieurs embranchements. L'écriture de cette structure est relativement complexe :

```
1 case expression in
2     pattern1)
3     # code
4     ;;
5     pattern2)
6     # code
7     ;;
8     *,*)
9     #code
10    ;;
11 esac
```

 Zsh

ex:

```

1  #! _bin_zsh -f
2  clear
3  echo -n "Bonjour $USER, de quel pays venez-vous ? : "
4  read COUNTRY
5
6  echo -n "En $COUNTRY, on parle le "
7
8  case $COUNTRY in
9      France | Belgique | Guadeloupe | Monaco )
10         echo -n "français"
11         ;;
12     Italie | "Saint Marin" )
13         echo -n "italien"
14         ;;
15     Espagne | Canaries )
16         echo -n "espagnol"
17         ;;
18     Allemagne )
19         echo -n "allemand"
20         ;;
21     *,*)
22         echo -n "anglais"
23         ;;
24 esac

```

10.11. Les structures itératives

10.11.1. La boucle while-do-done

La boucle `while` est utilisée pour exécuter un ensemble de commandes de manière répétitive **tant que la condition de sortie n'est pas atteinte**.

```

1  while condition
2  do
3      # code
4  done

```

Pour lire le contenu d'un fichier ligne par ligne, nous utiliserons cette syntaxe:

```

1  #! _bin_zsh -f
2  clear
3
4  file=_etc_passwd
5
6  while read -r line
7  do
8      echo $line
9  done < "$file"

```


10.11.2. La boucle `until-do-done`

La boucle `until` est utilisée pour exécuter un ensemble de commandes de manière répétitives **jusqu'à ce que la condition de sortie soit atteinte**. La condition est **inversée par rapport à la boucle `while`**

```
1 until condition
2 do
3     #code
4 done
```

 Zsh

10.11.3. La boucle `for-do-done`

La boucle `for` permet d'itérer sur les éléments d'une collection finie (un tableau, par exemple). C'est une des boucles les plus simples:

```
1 \#! /bin/zsh -f
2 for i in 1 2 3 4 5
3 do
4     echo "Tour de boucle n° \$i"
5 done
```

 Zsh

Comme pour toutes les autres structures, chaque élément de la boucle doit commencer sur une nouvelle ligne. Nous pouvons définir un *range* en employant la syntaxe `{x..y}` ou `{x..y..step}`.

I Annexes

II Exercices

III Powershell

Note

Pour éviter les messages d'erreurs dûs à des permissions non autorisées, vous pouvez utiliser le paramètre `-ErrorAction SilentlyContinue` à la fin de vos commandes.

1. Affichez l'aide de `Get-Help`.
2. Affichez l'aide sur le paramètre *PARAMETER* de `Get-Help`.
3. Affichez la liste des commandes Powershell.
4. Affichez l'aide sur le paramètre *SYNTAX* de `Get-Command`.
5. Affichez la **syntaxe** de `Get-Command`.
6. Affichez la **syntaxe** de `Get-Process`.
7. Affichez les commandes utilisant les *PSDrive*.
8. Déplacez-vous dans le répertoire `C:\Program Files`.
9. Déplacez-vous dans le répertoire **parent** de l'endroit où vous êtes.
10. Affichez l'**emplacement actuel**.
11. Ajoutez au *stack* « **Move** » votre emplacement actuel et passez dans `C:\WINDOWS`.
12. Ajoutez au *stack* « **Move** » votre emplacement actuel et passez dans `C:\Windows\System32`.
13. Affichez le contenu du *stack* « **Move** ».
14. Retournez au répertoire précédent.
15. Affichez **TOUS** les programmes de `C:\Windows`.
16. Affichez **TOUS** les dossiers (et **uniquement** les dossiers) à partir de `C:\`.
17. Affichez **TOUS** les dossiers système du disque `C:\`.
18. Affichez les fichiers *cachés* de votre **répertoire utilisateur**.
19. Affichez les fichiers **cachés non système** de l'ordinateur.
20. Affichez **TOUS** les fichiers de `C:\Windows` *SAUF* les programmes.
21. Affichez **TOUS** les programmes de `C:\` commençant par la lettre *A*.
22. Affichez les fichiers situés dans les *2 niveaux* de dossiers de **Appdata**.
23. Affichez **TOUS** les *fichiers systèmes/ commençant par /D*.
24. Affichez **tous** les fichiers *jpg* présents sur le disque `C:\`.

25. Créez un dossier à votre nom dans le répertoire **Documents**.
26. Dans ce dossier à votre nom, créez **en une commande** les dossiers *Toto, Tata, Tutu*.
27. Créez l'arborescence suivante dans le dossier à votre nom:

```
Toto > Musique
. > Images > Original
. > . > Travail
. > . > Jpg
. > Documents > Perso
. > . > Divers
```
28. Supprimez toute l'arborescence précédente en **une seule commande**. Demandez la confirmation à l'utilisateur.
29. Démarrez un **transcript** de votre session et sauvez-le sur votre bureau.
30. Affichez les imprimantes disponibles.
31. Quelles sont les commandes disponibles avec le verbe **move**?
32. Quels sont les alias qui commencent par la lettre **g**?
33. Affichez les informations du **disque** *c:* (le PSDrive C).
34. Affichez **uniquement** la date du système.
35. Affichez **uniquement** l'heure du système.
36. Quel jour était le 15 septembre 1512 ?
37. Créez un alias **who** qui affiche les utilisateurs locaux de l'ordinateur.
38. Créez un dossier **maison** dans le répertoire *Documents*.
39. Créez un raccourci vers **maison** sur votre *Bureau*.
40. Affichez les infos de l'ordinateur.
41. Testez si le chemin vers *C:\Windows\WinSxS* existe.
42. Simulez (avec *whatif*) et confirmez le vidage de la corbeille sur le disque *c:*.
43. Arrêtez votre **transcript**.
44. Écrivez un fichier *.log* avec comme contenu tous les fichiers avec l'extension *.log* se trouvant sur le disque dur.
45. **Comptez** le nombre de verbes utilisés dans les commandes Powershell.
46. **Comptez** le nombre de fichiers *.exe* présents sur le disque dur.
47. Copiez les images *jpg* de votre disque dur dans un dossier nommé *Backup/Images*.
48. Déplacez le dossier *Backup/Images* précédemment créé dans le dossier **Documents**.
49. Renommez les dossiers *Backup/Images* en **Bkp_Imgs**.
50. Supprimez le dossier **Bkp** et tout son contenu.
51. Écrivez « VIVE POWERSHELL » en noir sur fond jaune.
52. Affichez la liste des processus actifs commençant par la lettre **S**.
53. Indiquez **uniquement** le *displayname* et le *status* du service **fhsvc**.

54. Affichez les fichiers de Windows à partir de `C:\`, triés par taille croissante.
55. Affichez les fichiers de Windows à partir de `C:\`, triés par taille décroissante, en affichant **uniquement** la taille et le nom du fichier.
56. Écrivez un script qui demande un nom de fichier à l'utilisateur. Le script doit indiquer si le fichier existe ou non. (Aide: il faut tester le chemin...).
57. Écrivez un script qui demande un nom de fichier à copier, ainsi que l'endroit où le copier. Vérifiez que le fichier à copier existe et qu'il n'est déjà pas présent dans le dossier de destination.
58. Écrivez un script qui demande à l'utilisateur de deviner un nombre compris entre 0 et 100. L'ordinateur choisira ce nombre au hasard. L'ordinateur indiquera en vert **Plus grand** et en rouge **Plus petit**. Quand le nombre est trouvé, l'ordinateur affichera en couleur (au choix) le message **Bravo, vous avez trouvé *n* en *x* coups**, *n* et *x* étant à remplacer par leurs valeurs.
59. Améliorez le script précédent pour rejouer tant que l'utilisateur le souhaite et limiter le nombre d'essais à 10 coups.

IIII Linux

Note

Les exercices ont été réalisés avec zsh sous Fedora.

1. Affichez le manuel de la commande `man`.
2. Créez un fichier **toto** daté du *15 avril 2014*.
3. Créez un dossier **bröl** daté du *15 avril 2014*.
4. Affichez **en une seule commande** le contenu des dossiers `/var` et `/etc`.
5. Affichez un calendrier des 6 prochains mois.
6. Déplacez le fichier **toto** dans le dossier *bröl*.
7. Créez un lien **symbolique** de **bröl** dans le dossier *Bureau*.
8. Créez un lien **physique** du fichier **toto** dans le dossier *Bureau*.
9. Affichez les **15** premières lignes du fichier `/etc/dnsmasq.conf`.

10. Renommez le fichier *toto* en **toto.txt**.
11. Écrivez le contenu de l'historique des commandes dans *toto.txt*.
12. Supprimez le dossier **brol** avec tout son contenu. Faites en sorte de demander une confirmation de suppression.
13. Combien de lignes, de mots et de caractères comporte le fichier **/etc/services** ?
14. Combien de comptes utilisateurs sont définis sur le système? (Aide: voyez le fichier */etc/passwd*)
15. Affichez la liste des comptes utilisateurs (**uniquement le nom du compte!**) classée par ordre décroissant.
16. Affichez la date actuelle au format AAAAMMJJ.
17. Affichez l'heure actuelle.
18. Affichez le calendrier du mois de **septembre 1752**.
19. Quel est le type de fichier **/boot/efi/EFI/BOOT/BOOTX64.EFI** ?
20. Passez dans le répertoire **/usr/share/doc** , remontez dans le répertoire parent, puis indiquez via une commande dans quel répertoire vous vous trouvez.
21. Allez dans votre *répertoire personnel* **sans en taper le chemin**.
22. Allez dans le *répertoire précédent* **sans en taper le chemin**.
23. Listez tous les fichiers de votre *répertoire personnel*, même les fichiers cachés, sans afficher . et ...
24. Listez de façon détaillée le contenu de **/usr** sans changer de répertoire de travail.

25. Affichez l'arborescence des fichiers contenus dans **/var** , sans changer de répertoire de travail. (aide: *tree*)
26. Affichez de façon détaillée de le contenu du répertoire **/var/log** en classant les fichiers du plus vieux au plus récent.
27. Affichez de façon détaillée le répertoire **/home** sans lister son contenu.
28. Quel est le format des fichiers **/etc/passwd** , **/usr/bin/passwd** , **/bin/ls** et **/usr** ?
29. Affichez les informations contenues dans les **inodes** des fichiers précédents. (Aide: Regardez la commande *stat*).
30. Affichez le contenu du fichier **/etc/issue** . Que contient-il?
31. Affichez page par page le contenu du fichier **/etc/services** . Que contient-il?
32. Déterminez le format du fichier **/bin/false** et affichez son contenu avec la commande adéquate.
33. Affichez les chaînes de caractères contenues dans le fichier **/bin/false** .
34. Créez l'arborescence suivante dans votre **répertoire utilisateur**, sans changer de répertoire et avec le minimum de commandes:

```
maison
  > cave > vin
  > rdc  > cuisine
        > salon
  > etage > chambre 1
        > chambre 2
  > grenier
```
35. Créez en une seule commande l'arborescence **/tmp/rep1/rep2/rep3/rep4** .
36. Supprimez le répertoire **/tmp/rep1** . Est-ce possible?
37. Copiez le fichier **/etc/services** dans votre *répertoire utilisateur*.

38. Affichez **uniquement** le propriétaire du fichier précédemment copié.
39. Créez **en une seule commande** les fichiers vides suivants: *Chateau-Neuf du Pape*, *canapé*, *fauteuil*, *lit* et *armoire*.
40. Déplacez en effectuant le moins de commandes possibles: *Chateau-Neuf du Pape* dans le répertoire **Vin**, *canapé* et *fauteuil* dans le **Salon**, *lit* et *armoire* dans la **Chambre 1**.
41. Copiez les éléments de *Chambre 1* vers *Chambre 2*.
42. Dans le répertoire **Chambre 2**, renommez le fichier *lit* en *clic-clac*.
43. Affichez votre nom de connexion et votre UID.
44. Affichez les groupes auxquels vous appartenez.
45. Donnez les droits **r**, **w** et **x** aux autres utilisateurs sur le répertoire **Grenier**.
46. Créez le fichier caché *secret* dans le répertoire **Grenier**. Seul le propriétaire peut lire et écrire dans ce fichier.
47. Listez les processus lancés à partir de votre shell courant.
48. Listez tous les processus s'exécutant sur le système.
49. Listez tous les processus (démons inclus) en affichant l'identité sous laquelle ils s'exécutent.
50. Affichez la hiérarchie des processus s'exécutant sur le système. (Aide: `ps tree`)
51. Listez les signaux pouvant être envoyés aux processus. (Aide: `kill`)
52. Ouvrez le navigateur et tuez son processus avec son PID.
53. Ouvrez le navigateur et tuez son processus avec son nom.
54. Avec la commande `top`, ajoutez la colonne **PPID** dans l'affichage.
55. Avec la commande `top`, affichez uniquement les processus lancé par votre compte utilisateur.
56. Créez un compte utilisateur *rocky*, avec son nom complet *Rocky Lémoche*.

57. Créez le groupe **cours** et ajoutez *rocky* à ce groupe.
58. Créez le fichier *test* et faites en sorte que *rocky* et le groupe *cours* en soient propriétaires.
59. Comprimez le répertoire **/home** au format *bz2* via l'utilitaire *tar*.
60. Décompressez l'archive précédemment créée dans **/tmp/test** .
61. Recherchez le fichier *pacman.conf*.(Aide: *grep*)
62. Listez les fichiers de **/bin** en n'affichant **que les noms des répertoires**.
63. Listez les fichiers de **/bin** en n'affichant **que les noms de fichiers**.
64. Listez dans toute l'arborescence du système tous les fichiers supérieur à 10Mo. Supprimez les messages d'erreurs avec une redirection vers **/dev/null** .
65. Listez dans toute l'arborescence du système tous les fichiers ayant les droits **4755**.
66. Affichez le nom de tous les fichiers dont le nom commence par **p** dans **/etc** . Supprimez les messages d'erreurs avec une redirection.
67. Affichez le PID des processus *zsh* actuellement lancés sur le système. (Aide: *grep*).
68. Affichez toutes les lignes du fichier **/etc/services** contenant la chaîne « *http* ».
69. Affichez toutes les lignes du fichier **/etc/services** avec la chaîne « *http* » **en tant que mot**.
70. Affichez les lignes du fichier **/etc/passwd** ne contenant pas la chaîne de caractères « *home* ».
71. Combien de lignes du fichier **/etc/passwd** contiennent la chaîne de caractères « *sbin* »?
72. Quels fichiers du répertoire **/etc** contiennent votre nom d'utilisateur comme chaîne de caractères?
73. À quels numéros de lignes trouve-t-on votre nom d'utilisateur comme chaîne de caractères dans les fichiers précédents?
74. Affichez toutes les lignes du fichier **/etc/services** contenant la chaîne de caractères « *iana* », quelle que soit la casse des caractères.
75. Affichez **uniquement** le premier et le troisième champ du fichier **/etc/group** .
76. Triez le fichier **/etc/passwd** alphabétiquement suivant le login de chaque utilisateur.
77. Triez le fichier **/etc/passwd** numériquement suivant l'UID de chaque utilisateur.
78. Affichez une liste détaillée des fichiers présents dans **/etc** triés du plus grand au plus petit.

III Commandes principales Powershell avec leur équivalent Linux

Add-Content : echo ... >> *file*
Add-History : history
Clear-Content : echo << << >> *file*
Clear-History : history -c
Clear-Item : rm
Clear-Variable : unset *variable*
ConvertFrom-Csv : awk, sed
ConvertFrom-Json : jq
ConvertTo-Csv : sed
ConvertTo-Json : jq
ConvertTo-Xml : xsltproc
Copy-Item : cp
Disable-LocalUser : usermod -L *user*
Enable-LocalUser : usermod -U *user*
ForEach-Object : for *object* in *\$objects*; do *cmd*;
Format-Custom : awk, sed
Format-Hex : xxd
Format-List : ls -l
Format-Table : awk, sed
Format-Wide : awk, sed
Get-Alias : alias
Get-ChildItem : ls
Get-Command : which, type
Get-ComputerInfo : uname -a, lsb_release -a
Get-Content : cat
Get-Date : date
Get-FileHash : md5sum, sha1sum, sha256sum
Get-Help : man
Get-History : history
Get-Item : ls
Get-Job : ps
Get-LocalGroup : groups
Get-LocalUser : cat /etc/passwd
Get-Location : pwd
Get-Member : ls
Get-NetAdapter : lspci, lsusb
Get-NetIPConfiguration : ifconfig, ip addr show
Get-Process : ps
Get-Random : random
Get-Service : service, systemctl
Get-Uptime : uptime
Get-Variable : echo *\$var*
Get-Verb : pas de commandes
Group-Object : sort, uniq
Import-Csv : awk, sed
Import-Module : source

Invoke-Command : ssh, curl
Invoke-History : history
Invoke-Item : lancer le fichier
Join-Path : echo
Join-String : echo
Measure-Command : time
Measure-Object : wc
Move-Item : mv
New-Alias : alias
New-Item : touch
New-LocalGroup : groupadd
New-LocalUser : useradd
New-Module : module
New-Service : systemctl, service
New-Variable : variable=value
Out-Default : echo
Out-File : > *file*
Out-Host : echo
Out-Null : > /dev/null
Out-Printer : lpr
Out-String : echo
Pop-Location : popd
Push-Location : pushd
Read-Host : read
Remove-Alias : unalias
Remove-Item : rm
Remove-LocalGroup : groupdel
Remove-LocalGroupMember : gpasswd
Remove-LocalUser : userdel
Remove-Module : module
Remove-Service : systemctl, service
Remove-Variable : unset
Rename-Computer : hostnamectl set-hostname *new_name*
Rename-Item : mv
Rename-LocalGroup : groupmod
Rename-LocalUser : usermod
Restart-Computer : reboot
Restart-Service : service ... restart, systemctl
Resume-Service : service
Select-Object : awk, sed
Select-String : grep
Select-Xml : xmllint
Send-MailMessage : sendmail
Set-Alias : alias
Set-Content : echo *content* > *file*
Set-Date : date
Set-Item : chmod, chown

Set-LocalGroup : groupmod
Set-LocalUser : usermod
Set-Location : cd
Set-Service : systemctl
Set-Variable : variable=value
Sort-Object : sort
Split-Path : dirname, basename
Start-Process : nohup
Start-Service : service ... start, systemctl
Start-Sleep : sleep
Stop-Computer : shutdown
Stop-Process : kill
Stop-Service : service ... stop, systemctl
Stop-Service : systemctl
Suspend-Service : systemctl
Tee-Object : tee
Test-Connection : ping
Test-Path : test
Update-Help : mandb, man -f
Wait-Process : wait
Where-Object : grep
Write-Error : echo >&2
Write-Host : echo
Write-Output : echo