E-Concept Applications

L'informatique au service des professionnels

Introduction aux systèmes d'exploitation

E-Concept Applications
L'informatique au service des professionnels

SOMMAIRE

SOMMAIRE

1) Généralités

1.1) Historique de Linux

1.1) HISTORIQUE DE LINUX

- Né de la rencontre entre Unix et le monde des Logiciels Libres
- Noyau créé en 1991, diffusé sous licence libre GNU GPL à partir de 1992
- Son nom vient du créateur du noyau Linux : **Linus** Torvalds
- Répond aux spécifications de la norme POSIX
- Décliné rapidement sous forme de plusieurs distributions
- Exemples de distributions : Ubuntu, Debian, CentOS, Slackware, RedHat, etc.

1.2) Architecture globale de Linux

1.2) ARCHITECTURE GLOBALE DE LINUX

- Linux est avant tout un noyau, autour duquel va s'organiser l'ensemble du système d'exploitation (OS).
- Le noyau permet à l'OS de communiquer avec les différents composants matériels (processeur, mémoire, stockage, composants multimédias, etc.)
- Une distribution Linux est composée du noyau et d'un ensemble de paquets fournissant des outils et logiciels, formant un système d'exploitation complet et prêt à l'emploi.

1.2) ARCHITECTURE GLOBALE DE LINUX

Parmi les paquets fournis dans une distribution, on trouve généralement :

- Un terminal, permettant de manipuler l'OS en ligne de commande.
  Le terminal le plus répandu est « bash », également disponible sous Windows (en installant WSL) et Mac OS.
- Des pilotes pour la plupart des matériels existants
- Une interface graphique pour les systèmes à vocation bureautique
- Une sélection plus ou moins large et variée de logiciels selon la distribution.

1.3) Les fichiers

1.3) LES FICHIERS

- Dans un système Linux, absolument tout est représenté sous forme de fichiers.
- Il existe plusieurs types de fichiers :
  - Les fichiers réguliers : Fichiers « classiques », tels que documents, programmes, images, musique, etc.
  - Les répertoires : Equivalent des dossiers Windows. Fonctionnement identique, sauf pour la gestion des droits.
  - Les fichiers spéciaux : Liens, périphériques, canaux de communication & sockets.

1.3) LES FICHIERS

- L'ensemble du système est représenté sous forme d'une arborescence de fichiers.
- Il n'y a pas de lettre de lecteur comme Windows, même si plusieurs partitions peuvent être utilisées.
- Les partitions, comme les lecteurs externes (USB, optique, etc.) sont « montés » dans des répertoires de l'arborescence.

1.3) LES FICHIERS

- Les fichiers sont identifiés par un inode.
- L'inode contient entre autres les informations suivantes pour un fichier :
  - Le numéro d'inode
  - La taille du fichier
  - L'identifiant du périphérique contenant le fichier
  - L'identifiant du propriétaire du fichier, ainsi que celui du groupe
  - Le mode du fichier (ses droits d'accès)
  - L'horodatage du fichier (date de modification de l'inode, du fichier et de dernier accès)

1.3) LES FICHIERS

- Attention ! L'inode d'un fichier ne contient pas son nom !
- Un fichier peut en effet être présent à plusieurs endroits dans l'arborescence de fichiers, sous différents noms.
- Ce point sera abordé plus tard dans la partie sur les liens physiques et symboliques.

1.3) LES FICHIERS

- La racine du système est « / ».
- « / » sert également à séparer les répertoires dans le chemin de fichier.
- Par exemple, le fichier « `intro_système.txt` » situé dans le répertoire « `Documents` » de l'utilisateur « `mickael` » aura pour chemin d'accès : `/home/mickael/Documents/intro_système.txt`

1.4) Nommage des fichiers

1.4) NOMMAGE DES FICHIERS

- Un nom de fichier peut être composé de n'importe quel caractère alphanumérique et de la plupart des caractères spéciaux.
- Certains caractères spéciaux nécessitent d'être « échappés » quand on les utilise dans un nom de fichier en ligne de commande.
- Pour échapper un caractère, on le précède d'un « \ ». Par exemple « \$ ».
- Pour échapper le caractère « \ », on l'écrit simplement deux fois : « \\ ».

1.4) NOMMAGE DES FICHIERS

Les caractères interdits ou à éviter sont :

- « / » : Interdit, sert à séparer les noms de répertoire dans un chemin
- « \ » : Nécessite d'être « échappé », et peut poser des problèmes de compatibilité avec d'autres systèmes d'exploitation comme Windows. A éviter.
- « - » : Peut être utilisé, mais à éviter au **début** d'un nom de fichier.
  Le terminal interprète le « - » comme un paramètre pour une commande. Cela entrainerait donc une erreur de syntaxe avec un fichier nommé ainsi.

1.4) NOMMAGE DES FICHIERS

- « * », « ? », « : », « ' », « " », « # », « $ », « ; », « ! », « & » , « | », parenthèses « ( ) », accolades « { } », chevrons « <> », crochets « [ ] » : A éviter.
- Espace : Autorisé, mais nécessite d'être échappé.
- De façon générale, il est fortement recommandé de n'utiliser que les caractères alphanumériques (avec ou sans accents), le tiret « - », le souligné « _ », le point « . » et éventuellement l'espace (toutefois déconseillé).

1.4) NOMMAGE DES FICHIERS

Autres règles de nommage :

- Les noms de fichiers sont sensibles à la **casse**. Ainsi, « `bonjour.txt` » et « `Bonjour.txt` » sont deux fichiers **différents**.
- Un fichier commençant par un « . » est un fichier caché.
- Un nom de fichier est limité à 255 caractères. Il est toutefois conseillé de ne pas dépasser quelques dizaines de caractères au maximum, pour plus de lisibilité et de compatibilité avec d'autres systèmes d'exploitation.

1.4) NOMMAGE DES FICHIERS

- Le chemin d'accès d'un fichier peut être relatif ou absolu.
- Un chemin absolu se définit depuis la racine du système de fichiers « / » (cf exemple en partie 1.3, diapo 12.
- Un chemin relatif se détermine à partir du répertoire courant, en utilisant les noms spéciaux « . » et « . . »

1.4) NOMMAGE DES FICHIERS

- « . » représente le répertoire courant
- « . . » représente le répertoire parent.
- Exemple : vous êtes dans le répertoire « `/home/mickael/Documents/cours_1` » et vous voulez faire référence au fichier « `iut.png` » situé dans le répertoire « `/home/mickael/Images` »
  Vous pouvez écrire : « `../../Images/iut.png` »

1.4) NOMMAGE DES FICHIERS

- Autre exemple : Vous souhaitez accéder à un fichier portant le même nom qu'une commande, par exemple, « `screen` ».
- Pour y accéder, vous devez indiquer explicitement que vous souhaitez accéder au **fichier** « `screen` » du répertoire courant (et non la commande homonyme) en écrivant « `./screen` ».
- Une autre solution est d'appeler le fichier « `screen` » en utilisant son chemin d'accès absolu.

1.5) Les jokers

1.5) LES JOKERS

- Les caractères « `?` » et « `*` » sont appelés des jokers.
- « `?` » permet de remplacer un et un seul caractère (n'importe lequel) dans un nom de fichier.

  Exemple :

  « `co?plet` » correspond aussi bien à « `couplet` » qu'à « `complet` » ou n'importe quel fichier dont le nom commence par « `co` », suivi d'un caractère, suivi de « `plet` ».

1.5) LES JOKERS

- « * » permet de remplacer un nombre quelconque de caractères, **y compris aucun caractère**.

  Exemple :

  « `bon*.txt` » peut correspondre entre autre aux noms de fichiers suivants : « `bonjour.txt` », « `bonsoir.txt` », « `bon_à_tirer.txt` », « `bon_je_mets_un_nom_de_fichier_pour_l_exemple.txt` », ou encore « `bon.txt` », mais pas « `bonjour` », « `bon` » ou encore « `jour.txt` ».

1.5) LES JOKERS

- Exception : Le caractère « . » ne peut pas être remplacé par un joker s'il est au début d'un nom de fichier.
- Conséquence directe : une chaine avec joker porte soit sur les fichiers non-cachés, soit sur les fichiers cachés, mais pas les deux en même temps.

1.6) Les utilisateurs

1.6) LES UTILISATEURS

- Toute commande est exécutée par un utilisateur
- L'utilisateur « maître » est l'utilisateur nommé « root »
- Compte « root » à utiliser avec prudence : Il peut faire « ce qu'il veut » !
- Utilisateurs réunis par groupes
- Un utilisateur n'accède qu'aux fichiers dont lui ou son groupe est propriétaire et/ou a les droits nécessaires. Exception : « root », encore une fois.

1.6) LES UTILISATEURS

- Un compte utilisateur ne sert pas forcément à se connecter à une machine.
- Il peut être utilisé uniquement pour lancer certaines commandes de façon automatisée.

1.7) Droits des fichiers et répertoires

1.7) DROITS DES FICHIERS ET RÉPERTOIRES

- Chaque fichier ou répertoire possède des droits sur trois niveaux :
  - Le propriétaire
  - Le groupe (pas forcément un de ceux du propriétaire)
  - Les autres
- Chacun de ces trois niveaux peut posséder trois droits :
  - r : Droit de lecture
  - w : Droit d'écriture
  - x : Droit d'exécution

1.7) DROITS DES FICHIERS ET RÉPERTOIRES

- Droits représentés sous forme d'une chaine de caractères :
  « rwx rwx rwx »
- Chaque groupe de 3 lettres correspond à un niveau (dans l'ordre cité précédemment)
- Si un droit n'est pas attribué pour un niveau, on met un tiret à la place.

1.7) DROITS DES FICHIERS ET RÉPERTOIRES

Exemple

- Pour un fichier donné, le propriétaire a tous les droits, les utilisateurs membres du groupe du fichier ont le droit de lecture et d'écriture, et les autres ne peuvent que lire le fichier. Les droits s'écrivent :

rwx rw- r--

1.7) DROITS DES FICHIERS ET RÉPERTOIRES

- Représentation binaire : 1 pour un droit attribué, 0 pour un droit refusé, sur 3 bits.
- L'exemple précédent (rwx rw- r--) s'écrit donc :

  111 110 100
- En décimal, on convertit chaque groupe de 3 bits en décimal :

  764

1.7) DROITS DES FICHIERS ET RÉPERTOIRES

- Droits affichés par une commande : généralement préfixés par une lettre :
- « - » : Fichier ordinaire
- « d » : Répertoire
- « l » : Lien symbolique
- Autres lettres : Fichiers spéciaux. De façon générale : **Ne pas toucher !**
- **Note : Le droit d'exécution est nécessaire pour pouvoir lister le contenu d'un répertoire.**

2) Commandes de base de Linux

2) COMMANDES DE BASE DE LINUX

Ce qui est demandé de retenir :

- Le nom de chaque commande
- A quoi elles servent, indépendamment de leurs paramètres
- Savoir à quoi servent les principaux paramètres présentés ici en les voyant.

Ce qui n'est pas demandé :

- Savoir lister par cœur les paramètres et leur rôle.

2) COMMANDES DE BASE DE LINUX

- **Entrainez-vous à utiliser les commandes vues au quotidien !**
- Le temps gagné en ligne de commande est **conséquent** dès lors qu'on a un tout petit peu l'habitude de l'utiliser, par rapport à une interface graphique.
- Astuce en ligne de commande : la touche **tab** complète énormément de choses : Le nom d'une commande, d'un répertoire, d'un fichier, etc.
  Un double appui sur tab affichera la liste des possibilités s'il y en a plusieurs.

2.1) Syntaxe générale d'une commande

2.1) SYNTAXE GÉNÉRALE D'UNE COMMANDE

- Une commande s'appelle directement par son nom
- On peut spécifier un ou plusieurs paramètres à une commande.
- Les paramètres peuvent être soit directement une valeur (par exemple, un nom de fichier), soit un nom de paramètre précédé d'un « – » et suivi de la valeur correspondante.
- Par exemple :

```
ls -h /home/mickael
```

2.2) La documentation intégrée

2.2) LA DOCUMENTATION INTÉGRÉE

- Les systèmes Linux disposent généralement d'une documentation intégrée.
- Celle-ci est accessible via les commandes « man » et « whatis » et permet d'obtenir la documentation détaillée de la commande spécifiée en paramètre.
- La documentation d'une commande indique sa syntaxe générale, ainsi que le détail des paramètres possibles.
- Dans la syntaxe générale, si un paramètre est indiqué entre [ ], alors ce paramètre est facultatif.

2.2) LA DOCUMENTATION INTÉGRÉE

- « `man` »

  Toutes les commandes de linux disposent d'un manuel intégré. Pour le consulter, il vous suffit d'utiliser la commande « `man` ».

  - Syntaxe :

    `man [x] commande`

  - Paramètres :
    - `x` : Facultatif – Précise le n° de la page du manuel
    - `commande` : Commande dont vous souhaitez consulter le manuel

2.2) LA DOCUMENTATION INTÉGRÉE

- « `whatis` »

  « `whatis` » permet d'avoir un descriptif des différentes pages de manuels pour la commande donnée. Cela permet d'identifier la page pour « `man` » contenant l'information recherchée.

  - Syntaxe :

    `whatis` *`commande`*

  - Paramètres :

    - `commande` : nom de la commande sur laquelle porte la demande

2.2) LA DOCUMENTATION INTÉGRÉE

- Que fait la commande suivante ?

```
man man
```

- Elle affiche la documentation intégrée de la commande « `man` » !

2.3) Manipulation de fichiers

2.3) MANIPULATION DE FICHIERS

- « `pwd` »

  Indique le répertoire courant

  - Syntaxe :

    ```
    pwd
    ```

2.3) MANIPULATION DE FICHIERS

- « `cd` »

  Permet de changer de répertoire.

  - Syntaxe :

    `cd` *nom*

  - Rappel : Comme sous Windows et Mac, nous retrouvons deux répertoires spéciaux dans tous les répertoires :
    - « `.` » : Ce répertoire désigne le répertoire courant
    - « `..` » : Ce répertoire désigne le répertoire parent

2.3) MANIPULATION DE FICHIERS

- « `ls` »

  Permet de lister le contenu d'un répertoire.

  - Syntaxe :

    `ls` *`[options] [fichier]`*

  - options (liste partielle) :
    - -C : Affiche la liste en colonnes , triée verticalement
    - -R : Affiche récursivement le contenu des sous-répertoires

2.3) MANIPULATION DE FICHIERS

- -a : Affiche tous les fichiers, y compris ceux commençant par un « . » (Fichiers cachés)
- -l : Affiche également le type de fichier, les droits d'accès, le nom du propriétaire et du groupe, la taille en octet et l'horodatage (entre autres).
- -r : Inverse l'ordre de tri
- -t : Tri par date et non par ordre alphabétique (t comme « time »)
- -h : « Human readable » : Ajout une lettre pour l'unité de taille, comme k pour kilo-octet, plus lisible qu'en octets.

- fichier : Le(s) nom(s) de fichiers/répertoires à lister. Si plusieurs noms donnés, listera chacun des éléments donnés en paramètre. Si non spécifié, utilise le répertoire courant.

2.3) MANIPULATION DE FICHIERS

- « `mkdir` »

  Crée un répertoire

  - Syntaxe :

    `mkdir [options] repertoire`

  - options (liste partielle) :

    - -p : Crée les répertoires parents si nécessaire

      Exemple : mkdir -p a/b/c => Crée a et b en plus de c si nécessaire.

  - repertoire : Nom du répertoire à créer

2.3) MANIPULATION DE FICHIERS

- « `cp` »

  Gère la copie de fichiers

  - Syntaxe :

    `cp [options] source destination`

  - options (liste partielle) :

    - -f : Efface automatiquement les fichiers dans la destination si ceux-ci existent déjà. Attention ! Ne demande aucune confirmation ! A utiliser prudemment, surtout en combinaison avec -r.
    - -r : Copie récursivement tout le contenu des répertoires

2.3) MANIPULATION DE FICHIERS

- -H : Si un lien symbolique est spécifié dans les fichiers à copier, suit le lien plutôt que de copier le lien lui-même. Les liens rencontrés par une copie récursive (donc, non spécifiés directement dans les fichiers à copier) ne seront pas suivis.

- source : Le(s) nom(s) de fichiers/répertoires à copier (séparés par des espaces si plusieurs)
- destination : Nom du répertoire de destination ou nom de fichier à utiliser pour la copie (Permet ainsi de renommer un fichier en même temps que de le copier. Un seul fichier source possible, dans ce cas)

2.3) MANIPULATION DE FICHIERS

- IMPORTANT : **Les copies appartiennent à l'utilisateur qui a effectué la copie !** Si vous faites la copie en tant que « root » pour un autre utilisateur, pensez à rétablir le bon propriétaire et le bon groupe après la copie sur les fichiers copiés !

- « `mv` »

Gère le déplacement de fichiers

- Même syntaxe que « `cp` », à l'exception du paramètre -r qui n'a aucun sens avec cette commande.
- Note : Contrairement à « `cp` », « `mv` » **conserve** le propriétaire et le groupe des fichiers.

2.3) MANIPULATION DE FICHIERS

- « `rm` »

  Gère la suppression de fichiers

  - Syntaxe :

    `rm [options] fichier`

  - options (liste partielle) :

    - -f : Efface automatiquement les fichiers sans demander de confirmation.
      A utiliser très prudemment, particulièrement en combinaison avec -r.
      Par exemple : « `rm -rf /` » supprimera purement et simplement tous les fichiers du système

2.3) MANIPULATION DE FICHIERS

- -r : Efface récursivement tous les fichiers, y compris les fichiers des sous-répertoires et les sous-répertoires eux-mêmes
- fichier : Le(s) nom(s) de fichiers/répertoires à effacer

2.3) MANIPULATION DE FICHIERS

- « `rmdir` »

  Gère la suppression de répertoires

  - Syntaxe :

    `rmdir` *`[options] répertoire`*

  - options (liste partielle) :

    - -p : Efface également le répertoire parent si celui-ci devient vide après suppression du répertoire spécifié. Par exemple, `rmdir -p a/b/c` est équivalent à `rmdir a/b/c; rmdir a/b; rmdir a`

2.3) MANIPULATION DE FICHIERS

- repertoire : Le(s) nom(s) de répertoire(s) à supprimer.
- IMPORTANT :

Un répertoire doit être vide pour pouvoir être supprimé avec rmdir !!!

2.3) MANIPULATION DE FICHIERS

- « `touch` »

  Permet de créer un fichier (vide)

  - Syntaxe :

    ```
    touch [options] fichier
    ```

  - options :

    Pour les options de touch, référez-vous à l'aide de Linux (man).

  - fichier : Le nom du fichier à créer.

2.3) MANIPULATION DE FICHIERS

- « `more` »

  Permet d'afficher le contenu d'un fichier avec défilement

  - Syntaxe :

    `more [options] fichier`

  - options :

    Pour les options de more, référez-vous à l'aide de Linux (man). Aucune option particulière n'est nécessaire pour un usage de base.

  - fichier : Le nom du fichier à afficher.

2.3) MANIPULATION DE FICHIERS

- Principales actions disponibles une fois la commande lancée :
  - Appui sur la touche « Entrer » : Fait défiler le contenu du fichier d'une ligne
  - Appui sur la touche « Espace » : Fait défiler le contenu du fichier d'un écran
  - Appui sur la touche « q » : Arrête « more » sans afficher la fin du fichier
  - Pour les autres commandes, référez-vous à l'aide.
- D'autres commandes permettent d'afficher des fichiers, avec d'autres possibilités :
  - cat
  - less

Référez-vous à l'aide man pour plus de détails.

2.3) MANIPULATION DE FICHIERS

- « `find` »

  Permet de rechercher un fichier dans l'arborescence de fichiers

  - Syntaxe de base :

    `find` *`racine`* `-name` *`fichier`*

  - racine : Répertoire de départ de la recherche (La recherche est récursive).
  - fichier : Le nom de fichier à rechercher. Vous pouvez utiliser les jokers.
  - D'autres possibilités de recherches et d'autres options pour find existent. Plus d'infos : http://doc.ubuntu-fr.org/recherche_ligne_commande

2.3) MANIPULATION DE FICHIERS

- « `wget` »

  Permet de télécharger un fichier

  - Syntaxe de base :

    `wget [options] [URL]`

  - options (liste partielle) :

    - -b : Passe la commande en arrière-plan pour libérer le terminal
    - -O *fichier* : Spécifie le nom du fichier de destination. S'il y a plusieurs fichiers téléchargés, ils sont concaténés dans le fichier spécifié. Attention ! Si le fichier existe, il est remplacé.
      Si l'option n'est pas indiquée, le nom d'origine du fichier est utilisé, et si plusieurs fichiers sont téléchargés, ils sont tous enregistrés dans des fichiers distincts.

2.3) MANIPULATION DE FICHIERS

- -i *fichier* : permet de télécharger toutes les URL présentes dans *fichier* (au format texte, une URL par ligne).

- URL : L'URL du fichier à télécharger (inutile avec l'option -i)

- Notes :
  wget est un outil puissant qui permet de faire bien plus que télécharger un simple fichier. Vous pouvez également télécharger une page web, voire un site web entier. Cf. man wget.
  Plus d'infos également sur https://doc.ubuntu-fr.org/wget

2.3) MANIPULATION DE FICHIERS

- « Nano » :
  Nano est un éditeur de texte en ligne de commande très complet et assez simple d'utilisation.
- Les principales commandes sont indiquées en bas de l'écran, à utiliser avec la touche Ctrl.
- Une présentation assez complète est disponible à l'adresse suivante :
  http://openclassrooms.com/courses/reprenez-le-controle-a-l-aide-de-linux/nano-l-editeur-de-texte-du-debutant

2.4) Gestion des droits

2.4) GESTION DES DROITS

- « `chmod` »

  Permet de modifier les permissions sur des fichiers/répertoires

  - Syntaxe de base :

    ```
    chmod [options] mode fichier
    ```

  - options (liste partielle) :

    - -R : Applique la modification de façon récursive

2.4) GESTION DES DROITS

- mode : Peut être spécifié soit en décimal, soit sous la forme suivante : x[+/-/=]y
  Dans le second cas :
  - x correspond au niveau auquel doit s'appliquer la modification : u pour le propriétaire du fichier, g pour le groupe, o pour tout les autres. Si aucune lettre n'est spécifiée, la modification s'appliquera aux trois niveaux en même temps. Si deux lettres sont spécifiées, s'appliquera aux deux niveaux concernés.
  - [+/-/=] : Indique respectivement s'il faut ajouter un droit, retirer un droit ou utiliser exclusivement le droit spécifié.
  - y : la lettre correspondant au droit à modifier (r, w ou x)

2.4) GESTION DES DROITS

- Exemples :
  - chmod 640 mon_fichier autorisera la lecture et l'écriture sur le fichier pour son propriétaire, la lecture seule pour les membres du groupe, et aucun droit d'accès pour les autres utilisateurs.
  - chmod u+x mon_fichier ajoutera le droit d'exécution au propriétaire.
  - chmod ug-r mon_fichier retirera le droit de lecture au propriétaire ET aux membres du groupe
  - chmod =r mon_fichier n'autorisera que le droit de lecture pour tout le monde (propritaire, membres du groupe et autres utilisateurs).

2.4) GESTION DES DROITS

- « `chown` »

  Permet de modifier le propriétaire et le groupe d'un fichier/répertoire

  - Syntaxe de base :

    `chown [options] propriétaire[:groupe] fichier`

  - options (liste partielle) :

    - -R : Applique la modification de façon récursive

2.4) GESTION DES DROITS

- propriétaire : Nom de l'utilisateur qui doit devenir propriétaire
- :groupe : spécification du groupe (facultatif)
- fichier : Fichier(s) concernés par la modification.

2.5) Personnaliser son terminal

2.5) PERSONNALISER SON TERMINAL

- « `alias` »

  Permet de créer un alias d'une commande

  - Syntaxe de base :

    ```
    alias nom_alias='commande'
    ```

  - nom_alias : Nom que vous souhaitez donner à votre alias.
  - commande : la commande que vous souhaitez mettre en alias, avec ses éventuels paramètres.

2.5) PERSONNALISER SON TERMINAL

- Exemple :

```
alias ls='ls -color=auto'
alias ll='ls -l'
```

➔ Dans ce second cas, cela équivaut à `ls -color=auto -l` !

- Mémoriser les alias pour qu'ils soient actifs à l'ouverture du terminal :
  placez-les dans le fichier ~/.bashrc, à la fin.
  ATTENTION ! Vérifiez bien la syntaxe, en cas d'erreur, vous planterez votre terminal !
  Essayez de vous loguer sur un autre terminal avant de fermer celui en cours !

2.5) PERSONNALISER SON TERMINAL

- Vous pouvez aussi personnaliser le texte affiché au début de chaque ligne (le « prompt »). Plus d'infos sur https://www.malekal.com/personnaliser-invite-prompt-terminal-linux/
- ATTENTION ! Là aussi, vérifiez bien la syntaxe dans votre .bashrc !

2.6) Communication entre commandes et filtrage

2.6) COMMUNICATION ENTRE COMMANDES ET FILTRAGE

- Sous linux, une commande affiche généralement ses informations directement dans le terminal et obtient les informations dont elle a besoin du clavier.
- Ces canaux de communication peuvent être modifiés, et même transmis entre plusieurs commandes.
- On distingue 3 canaux : entrée standard, sortie standard et sortie d'erreur.

2.6) COMMUNICATION ENTRE COMMANDES ET FILTRAGE

- Entrée standard (stdin) :
  On appelle entrée standard le moyen par lequel des données sont envoyées à une commande. Par défaut, il s'agit du clavier.
- Sortie standard (stdout, canal de sortie n°1) :
  On appelle sortie standard le moyen par lequel sont délivrées les informations envoyées par une commande. Par défaut, c'est le terminal.
- Sortie d'erreur (stderr, canal de sortie n°2) :
  On appelle sortie d'erreur le moyen par lequel sont délivrés les messages d'erreur d'une commande.

2.6) COMMUNICATION ENTRE COMMANDES ET FILTRAGE

- « `head` »

  Permet d'afficher les premières lignes d'un fichier ou de l'entrée standard

  - Syntaxe de base :

    ```
    head [options] [fichier]
    ```

  - options (liste partielle) :
    - -n *x* : x = nombre de lignes à afficher. 10 si -n non précisé
    - fichier : Le fichier à traiter. Si non-précisé, l'entrée standard est utilisée

2.6) COMMUNICATION ENTRE COMMANDES ET FILTRAGE

- « `tail` »

  Permet d'afficher les dernières lignes d'un fichier ou de l'entrée standard

  - Syntaxe de base :

    ```
    tail [options] [fichier]
    ```

  - options (liste partielle) :
    - -n *x* : x = nombre de lignes à afficher. 10 si -n non précisé
    - fichier : Le fichier à traiter. Si non-précisé, l'entrée standard est utilisée

2.6) COMMUNICATION ENTRE COMMANDES ET FILTRAGE

- « `grep` »

  Permet de rechercher les lignes d'un fichier ou de l'entrée standard contenant une expression donnée

  - Syntaxe de base :

    ```
    grep [options] pattern [fichier]
    ```

  - pattern : expression à rechercher dans la ligne. Il s'agit d'une expression régulière. (Une chaine simple fonctionne et sera traitée comme un mot entier)
  - fichier : Le fichier à traiter. Si non-précisé, l'entrée standard est utilisée

2.6) COMMUNICATION ENTRE COMMANDES ET FILTRAGE

- options (liste partielle) :
  - -i : Ignore la casse
  - -v : Permet de rechercher les lignes qui ne correspondent PAS à pattern
  - -x : Permet de rechercher les lignes qui correspondent exactement à pattern, et non qui contiennent pattern
  - -q : Exécute la commande sans rien afficher à l'écran. Renvoie 0 si une correspondance est trouvée.

Note : Pour rechercher « toto » au début de la ligne, utilisez "^toto", pour le rechercher à la fin de la ligne, utilisez "toto$"

2.6) COMMUNICATION ENTRE COMMANDES ET FILTRAGE

- « `sort` »

  Permet de trier les lignes d'un fichier ou de l'entrée standard

  - Syntaxe de base :

    ```
    sort [options] [fichier]
    ```

  - options (liste partielle) :
    - -n : Trie en tenant compte de la valeur numérique des chaines de caractères
    - -h : des nombres dans un format lisible par un humain (par exemple : 2K, 5M, 1G)
    - -r : inverse l'ordre de tri
    - -f : Ignore la casse (= ne tient pas compte des minuscules/majuscules)

2.6) COMMUNICATION ENTRE COMMANDES ET FILTRAGE

- « `cut` »

  Permet de découper des lignes de texte selon un séparateur donné

  - Syntaxe de base :

    ```
    cut [options] [fichier]
    ```

  - fichier : Si spécifié, utilise les lignes du fichier, sinon, les lignes de l'entrée standard
  - options (liste partielle) :
    - -d*x* : définit le délimiteur « *x* » entre chaque élément de la ligne de texte.

      Par exemple : « -d: » spécifie le symbole « : » comme délimiteur

2.6) COMMUNICATION ENTRE COMMANDES ET FILTRAGE

- -fx : Indique le champ « *x* » à afficher. « -f2 » indique le 2nd champ. On peut indiquer plusieurs champs séparés par des virgules : « -f2,5 » affiche le 2e et le 5e champ.
  Note : si une ligne ne contient pas le délimiteur spécifié par « -d », elle sera quand même affichée.
- -s : Permet de ne pas afficher les lignes ne contenant pas le délimiteur spécifié par « -d »

2.6) COMMUNICATION ENTRE COMMANDES ET FILTRAGE

- Les tubes (pipe) permettent de transmettre la sortie standard d'une commande à l'entrée standard d'une autre
- Syntaxe : *commande_1 | commande_2*
- Exemple : Afficher les 30 dernières lignes du log de démarrage du système :

```
more /var/log/boot.log | tail -n 30
```

2.6) COMMUNICATION ENTRE COMMANDES ET FILTRAGE

- Redirection de la sortie standard (canal n°1) vers un fichier :
  `macommande > mon_fichier` → Crée un nouveau fichier à chaque fois !
  `macommande >> mon_fichier` → Ajoute à la fin d'un fichier
- Redirection de la sortie d'erreur (canal n°2)
  `macommande 2> error_log` (ou `macommande 2>> error_log`)
- Redirection de la sortie d'erreur sur la sortie standard :
  `macommande 2>&1`

2.6) COMMUNICATION ENTRE COMMANDES ET FILTRAGE

- Note :
  `macommande > mon_fichier` **et** `macommande 1> mon_fichier` **sont équivalents**

- Redirection de l'entrée standard :
  `macommande < monfichier`
- Exemple :
  `cat < monfichier` **est équivalent à** `cat monfichier`

2.6) COMMUNICATION ENTRE COMMANDES ET FILTRAGE

- Afficher les logins utilisateurs commençant par "ro"
  Les logins utilisateurs sont dans la 1ère colonne du fichier "/etc/passwd" (avec le séparateur ":")

```
cat /etc/passwd | cut -d: -f1 | grep '^ro'
```

- Afficher les lignes contenant le mot "error" parmi les lignes 60 à 100 du fichier toto.txt

```
head -n 100 toto.txt | tail -n 40 | grep error
```

ou

```
cat toto.txt | head -n 100 | tail -n 40 | grep error
```

2.6) COMMUNICATION ENTRE COMMANDES ET FILTRAGE

- Extraire les lignes 5 à 13 du fichier data.txt
  Rechercher les lignes contenant le mot toto.
  Enregistrer le résultat dans un fichier toto.txt

```
head -n 13 data.txt | tail -n 8 | grep toto > toto.txt
```

- **Que fait :** `MaCommande toto.txt 2>&1 >> titi.txt < tutu.txt`

Exécute MaCommande avec le paramètre « toto.txt », envoie sur l'entrée standard le contenu de tutu.txt, redirige la sortie d'erreur sur la sortie standard (actuellement l'écran), puis la sortie standard dans le fichier titi.txt
Note : la sortie d'erreur reste sur l'écran ! L'ordre des redirections est important !

3) Architecture de linux et du système de fichiers

3.1) Noyau, démarrage, processus

3.1) NOYAU, DÉMARRAGE, PROCESSUS

- Linux est en premier lieu un noyau (= kernel), avant d'être un système d'exploitation.
- Le **noyau** gère les communications avec le matériel : CPU, RAM, disques de stockage, composants multimédias (son, image, etc.), réseau, systèmes d'entrées (Clavier, souris, etc.), et bien d'autres encore.
- Dans le cas où le noyau ne sait pas dialoguer avec un matériel spécifique, on installera un **pilote** qui apportera au système les éléments nécessaires au dialogue avec le périphérique concerné.

3.1) NOYAU, DÉMARRAGE, PROCESSUS

- L'ensemble du système d'exploitation (OS) est structuré sur un schéma modulaire en étoile, dont le centre est le noyau.
- Conséquence directe : le noyau est le premier élément chargé au démarrage du système. Sans lui, rien ne fonctionne.
- L'architecture modulaire de Linux permet à de nombreux éléments de « planter » sans impact fatal sur l'exécution de l'OS. Au pire, ce dernier détruit le processus concerné, puis le recharge.

3.1) NOYAU, DÉMARRAGE, PROCESSUS

- L'architecture modulaire de l'OS permet également d'installer et de mettre à jour de nombreux paquets sans avoir besoin de redémarrer le système.
- Par contre, le noyau doit être extrêmement fiable ! Si celui-ci vient à planter, c'est toute la machine qui plante. C'est ce qu'on appelle un « Kernel Panic ».
- La seule solution pour sortir d'un Kernel Panic est de redémarrer physiquement la machine.

3.1) NOYAU, DÉMARRAGE, PROCESSUS

- La séquence de démarrage d'un système Linux est la suivante :

1. Le chargeur d'amorçage (généralement « GRUB2 ») est lancé par le BIOS (ou l'UEFI) de la machine
2. Le chargeur d'amorçage charge ensuite le noyau, qui monte le système de fichier racine (« / »).
3. Le noyau démarre alors le premier processus de l'OS : « init », dont le rôle est de charger tous les éléments de l'OS, et le second processus (sur les systèmes assez récents) : « kthread », qui chargera les différents modules du noyau.

3.1) NOYAU, DÉMARRAGE, PROCESSUS

- Sous Linux, chaque programme lancé est appelé un processus.
- Chaque processus est identifié par un PID : « Processus IDentifier »
- Un processus comporte également deux autres informations : l'UID de l'utilisateur qui l'a lancé, et le PID de son processus parent (PPID)
- Init porte toujours le PID 1, puisque c'est le premier à être chargé. Le n°2 sera kthread. Ce sont les deux seuls processus dont le PPID est 0.

3.1) NOYAU, DÉMARRAGE, PROCESSUS

- « `ps` »

  Affiche la liste des processus lancés dans l'invite de commande en cours

  - Syntaxe de base :

    `ps [options]`

  - Options en syntaxe BSD (liste partielle) :
    - -a : Affiche les processus de tous les utilisateurs fonctionnant dans un terminal
    - -ax : Affiche l'intégralité des processus du système. Équivalent à ps -A en syntaxe traditionnelle.

3.1) NOYAU, DÉMARRAGE, PROCESSUS

- -u : Affiche en plus le nom de l'utilisateur du processus et diverses informations (Conso CPU & RAM, date de lancement, etc.). Peut-être combiné avec -a et -ax : `ps -au` ou `ps -aux`
- Options en syntaxe traditionnelle (liste partielle) :
  - -e ou -A : Affiche tous les processus (les deux sont identiques)
  - -f : Affiche des informations supplémentaires, notamment l'utilisateur et le PPID.
  - -H : Trie les processus de façon hiérarchique. Permet de visualiser facilement qui a lancé qui dans les processus.

    Note : Les options sont également combinables pour simplifier l'écriture : `ps -efH`

3.1) NOYAU, DÉMARRAGE, PROCESSUS

- « `kill` »

  Permet d'arrêter un processus

  - Syntaxe de base :

    `kill` *`signal pid`*

  - Signal (liste partielle) :
    - -TERM ou -15 (ou sans signal spécifié) : Demande l'arrêt normal du processus.
    - -INT ou -2 : Arrête le processus, par interruption et non par arrêt normal.

3.1) NOYAU, DÉMARRAGE, PROCESSUS

- -KILL ou -9 : Force l'arrêt du processus, y compris si celui-ci ne répond plus. A n'utiliser qu'en cas de plantage du processus concerné.
- pid : PID du processus, visible notamment grâce à la commande « ps »

Effectuez toujours un arrêt avec les signaux les plus faibles (dans l'ordre ci-dessus) !

- « `killall` »

Même fonctionnement que « kill », sauf que « killall » prend le nom du processus en paramètre, et non son PID.

A noter que si le processus est lancé plusieurs fois, toutes les occurrences sont concernées !

3.2) Arborescence générale de Linux

3.2) ARBORESCENCE GÉNÉRALE DE LINUX

- Ensemble du système représenté sous forme d'une arborescence de fichiers.
- Pas de lettre de lecteur comme Windows, même si plusieurs partitions peuvent être utilisées.
- Les partitions, comme les lecteurs externes (USB, optique, etc.) sont « montés » dans des répertoires de l'arborescence.
- La racine du système est « / »

3.2) ARBORESCENCE GÉNÉRALE DE LINUX

- Principaux répertoires (Descriptions simplifiées) :

`/` → Racine du système

`/bin/` → Contient les commandes de base. Modifications déconseillées. (bin = **bin**aire, dans le sens « programme compilé »)

`/boot/` → Contient les fichiers nécessaires au démarrage (généralement GRUB2). Modifications déconseillées.

3.2) ARBORESCENCE GÉNÉRALE DE LINUX

`/dev/` → Représente le matériel de la machine et permet à Linux de le manipuler. **NE SURTOUT PAS TOUCHER !** (=**Dev**ice files)

`/etc/` → La plupart des fichiers de configuration du système et des logiciels installés. A moins d'être sûr de ce que vous faites, évitez de modifier les fichiers situés dans ce répertoire. Une erreur de configuration pourrait planter en partie ou en totalité le système.

3.2) ARBORESCENCE GÉNÉRALE DE LINUX

`/home/` → Contient les données des utilisateurs. Ce répertoire est généralement monté sur une partition spécifique. Pas forcément utilisé dans des usages spécifiques, ou sous un autre nom (serveur web géré par certaines interfaces d'administration, par exemple)

`/lib/`, `/lib32/`, `/lib64/` → Contient les **lib**rairies (aussi appelées bibliothèques, équivalent des DLL de Windows)

`/lost+found/` → Equivalent du répertoire found.000 de Windows : Contient les fichiers récupérés suite à des corrections du système de fichier

3.2) ARBORESCENCE GÉNÉRALE DE LINUX

`/media/` → Répertoire dans lequel sont montés « automatiquement » les lecteurs (**media**s) supplémentaires (sauf config. spécifique), tels que disques durs et partitions autres que celles utilisées spécifiquement par le système, ainsi que les lecteurs externes.

`/mnt/` → Idem /media/. Dépend des distributions. (mnt = **m**ou**nt**ed filesystems)

`/opt/` → Contient généralement les logiciels commerciaux ou non-inclus avec le système (opt = **opt**ional softwares)

3.2) ARBORESCENCE GÉNÉRALE DE LINUX

`/proc/` → Contient les informations sur l'état du système et les différents **proc**essus en fonction. Modifications déconseillées

`/root/` → Répertoire de l'utilisateur « **root** », le seul à ne pas être dans `/home/`

`/sbin/` → Contient les commandes (**bin**aires) d'administration **s**ystème (vérification et réparation des disques, mise en place du réseau, etc.)

`/tmp/` → Contient les fichiers **tmp**oraires du système

3.2) ARBORESCENCE GÉNÉRALE DE LINUX

`/usr/` → Contient généralement les logiciels fournis avec le système et utilisés par les utilisateurs (**us**e**r**s)

`/var/` → Répertoire de stockage de données. Assimilable au répertoire « Documents publics » de Windows. (var fait référence à des données **var**iables, contrairement aux binaires qui sont globalement statiques)

Note : En fonction des distributions de linux, certains répertoires peuvent être absents, et d'autres peuvent figurer en plus.

3.3) Les démons

3.3) LES DÉMONS

- Démons = Services sous Microsoft Windows
- Lancés généralement par « init » au démarrage de la machine. Peut parfois être lancé sur un événement précis.
- Fonctionnent en arrière-plan
- Pas besoin de session utilisateur ouverte
- Contrairement à Windows où il faut souvent redémarrer le PC en cas de modification d'un service, sous Linux, il suffit de redémarrer le démon.

3.3) LES DÉMONS

- 1ère méthode de contrôle des démons : Par script init.d
- Syntaxe :
  `/etc/init.d/nom_du_service commande`
- nom_du_service : Nom du service à manipuler
- commande : start, stop ou restart. D'autres commandes peuvent être disponibles. Dans ce cas, appelez le script sans commande pour les détails.

3.3) LES DÉMONS

- 2ème méthode de contrôle des démons : Commande service
- Syntaxe :

  `service nom_du_service commande`

  nom_du_service : Nom du service à manipuler
- commande : start, stop, restart, --status-all (sans nom de service)

3.3) LES DÉMONS

- 3ème méthode de contrôle des démons : Commande systemctl (du gestionnaire « systemd »)
- Syntaxe :
  `systemctl commande nom_du_service`
  nom_du_service : Nom du service à manipuler (éventuellement avec l'extension « .service »)
- commande : start, stop, restart, reload (recharge la config sans redémarrer le processus, si le programme le prend en charge), reload-or-restart, enable, disable, status

3.3) LES DÉMONS

- Tous les démons ne sont pas forcément gérables par toutes les méthodes !
- Les méthodes ne sont pas forcément toutes disponibles selon les distributions
- IMPORTANT : Pensez à redémarrer un démon si vous changez sa configuration pour qu'il prenne en compte les modifications !!!
- Certains démons nécessitent d'être arrêtés **avant** de modifier leur fichier de config, car ils l'enregistrent lors de leur arrêt !

3.3) LES DÉMONS

- Exemple : Je souhaite modifier la configuration du serveur web Apache de ma machine :

1. Je modifie le ou les fichiers de configuration concernés dans le répertoire `/etc/apache2/`
2. Je redémarre le démon apache2, avec au choix :
   `/etc/init.d/apache2 restart`
   `service apache2 restart`
   `systemctl restart apache2` (Généralement le plus utilisé)

3.4) Visualiser les informations systèmes

3.4) VISUALISER LES INFORMATIONS SYSTÈMES

- « `date` »

  Permet d'afficher et de configurer la date et l'heure du système.

  - Syntaxe de base :

    `date` *`[options]`*

  - Sans option : Affiche la date, l'heure et le fuseau horaire définis sur le système
  - options (liste partielle) :
    - +FORMAT : Permet de contrôler la façon d'afficher la date et l'heure, en remplaçant « FORMAT » par un code défini.

3.4) VISUALISER LES INFORMATIONS SYSTÈMES

- Le format « %s » (commande complète : « `date +%s` ») permet d'afficher un **« timestamp » (nombre de secondes écoulées depuis le 1er janvier 1970).** Cf « man » pour le détail des formats.
- -d *chaine* : Affiche la date indiquée dans *chaine* plutôt que la date du système.
  chaine peut être une date au format inversé « Année/Mois/Jour » ou au format américain « Mois/Jour/Année », suivi éventuellement d'une heure : heures:minutes:secondes (les secondes sont facultatives). Par exemple : `date -d "11/06/2023 13:30"`
  ou encore : `date -d « 2023/11/06"`
  la chaine peut également être un timestamp, précédé d'un « @ » :
  `date -d @1699273800`
  Pour plus de détails et pour configurer la date : `man date`

3.4) VISUALISER LES INFORMATIONS SYSTÈMES

- « `uname` »

  Permet d'afficher les informations du noyau et de l'architecture matérielle

  - Syntaxe de base :

    `uname` *`[options]`*

  - Sans option : Equivalent à « uname -s »
  - options (liste partielle) :
    - -a : Affiche toutes les options. Equivalent à « uname –snrvmpio »
    - -s : Affiche le nom du noyau (Kernel) de l'OS.

3.4) VISUALISER LES INFORMATIONS SYSTÈMES

- -n : Affiche le nom d'hôte réseau.
- -r : Affiche la révision du noyau (Kernel release). Cette information vous permet notamment de savoir si vous fonctionnez en 32 ou en 64 bits (en général, se termine respectivement par « -32 » et « -64 »)
- -v : Affiche la version du noyau
- -m : Affiche le nom de l'architecture de la machine (par exemple, « i686 » pour des machines 32bits compatibles avec les architectures des Pentium 3, 4, et autres processeurs plus récents)

3.4) VISUALISER LES INFORMATIONS SYSTÈMES

- -p : Affiche le type de processeur (Par exemple, « i686 » pour un processeur 32bits compatible Pentium 3, 4 ou plus récent. Aujourd'hui, à peu près tous les processeurs sont « i686 » s'ils sont exploités en 32 bits)
- -i : Affiche la plate-forme matérielle (Par exemple, « i386 », pour indiquer une plateforme 32bits compatible Intel, soit la majorité des systèmes actuels quand ils sont exploités en 32 bits.)
- -o : Affiche le nom de l'OS (Généralement, « GNU/Linux »)

3.4) VISUALISER LES INFORMATIONS SYSTÈMES

- « df »

  Affiche la liste des partitions, l'espace occupé/disponible et les inodes

  - Syntaxe de base :

    df *[options]*

  - options (liste partielle) :

    - -h : Affiche l'espace disponible avec différentes unités (k pour kilo-octets, M pour Mega-octet, G pour Giga-octet, etc.) plutôt qu'en octets

3.4) VISUALISER LES INFORMATIONS SYSTÈMES

- -i : Affiche le nombre d'inodes total/utilisés/restant sur chaque partition. Un inode est l'identifiant unique de chaque fichier et répertoire d'une partition. Le nombre d'inode est limité par partition. Pour comparaison, dans une partition NTFS, l'équivalent de l'ensemble des inodes est la MFT.

3.4) VISUALISER LES INFORMATIONS SYSTÈMES

- « du »

  Affiche la quantité d'espace disque utilisée par chacun des arguments, et pour chaque sous-répertoire des répertoires indiqués en argument.

  - Syntaxe de base :

    du *[options] fichier*

  - options (liste partielle) :

    - -h : Affiche l'espace disponible avec différentes unités (k pour kilo-octets, M pour Mega-octet, G pour Giga-octet, etc.) plutôt qu'en kilo-octets

3.4) VISUALISER LES INFORMATIONS SYSTÈMES

- -s : Affiche seulement le total pour chaque argument « fichier »
- fichier : le répertoire ou le fichier à analyser. Vous pouvez utiliser les jokers.

3.4) VISUALISER LES INFORMATIONS SYSTÈMES

- « free »

  Affiche la quantité de mémoire libre/utilisée.

  - Syntaxe de base :

    free *[options]*

  - options (liste partielle) :
    - -b : Affiche la mémoire en bytes
    - -k : Affiche la mémoire en kilo octet
    - -m : Affiche la mémoire en méga octet

3.4) VISUALISER LES INFORMATIONS SYSTÈMES

- -g : Affiche la mémoire en giga octet
- -h : Choisit automatiquement l'unité la plus adaptée
- -s : Spécifie le délai de réaffichage de la mémoire. Si non-spécifié, n'affiche qu'une seule fois le résultat.
- -t : Affiche la ligne des totaux

- Exemple :

  free -m -s 5

  Affiche la mémoire du système en Mo toutes les 5s.

3.4) VISUALISER LES INFORMATIONS SYSTÈMES

- « top »

  Permet d'afficher en continu la liste des processus en cours, classés par ordre de consommation CPU, les informations sur la mémoire, etc.

  - Syntaxe de base :

    top

  - Commandes disponibles dans top :
    - ? : Affiche l'ensemble des commandes disponibles
    - f ou F : Permet de choisir les colonnes, leur ordre et l'ordre de tri.

3.4) VISUALISER LES INFORMATIONS SYSTÈMES

- Plus d'informations sur la signification des valeurs affichées ici : http://quick-tutoriel.com/296-utilisation-de-la-commande-top-et-notion-de-load-average-sous-linux/

3.4) VISUALISER LES INFORMATIONS SYSTÈMES

- « watch »
  Permet d'exécuter continuellement une commande à intervalles réguliers
  - Syntaxe de base :
    watch –n *t [--color] [--differences] 'commande'*
  - Paramètres :
    - -n *t* : t = nombre de seconde entre chaque exécution de la commande
    - *--color : Active la prise en charge des couleurs pour les commandes exécutées*
    - --differences : Mets en surbrillance les caractères qui ont changé entre deux affichage
    - '*commande*' : Commande à répéter, entre apostrophes
  - Exemple : watch -n 1 'df –h' affiche toute les secondes l'occupation des partitions

4) Gestion des utilisateurs

4.1) Utilisateurs et groupes

4.1) UTILISATEURS ET GROUPES

- Tout entité (personne physique ou programme particulier) devant interagir avec un système Linux est authentifiée sur cet ordinateur par un utilisateur.
- Chaque utilisateur est identifié par un nom unique et un ID unique (UID).
- Le compte root, en tant que « super-utilisateur », possède toujours l'UID 0.

4.1) UTILISATEURS ET GROUPES

- Chaque utilisateur fait partie d'un ou plusieurs groupes. Si aucun groupe n'est spécifié à la création de l'utilisateur, un groupe du même nom que ce dernier est créé.
- Chaque groupe est également identifié par un numéro unique : le GID
- Les opérations de création, modification et suppression d'utilisateurs ne peuvent se faire qu'en étant « root ».

4.2) Lister les utilisateurs et les groupes

4.2) LISTER LES UTILISATEURS ET LES GROUPES

- L'ensemble des utilisateurs et de différentes informations les concernant sont présents dans le fichier /etc/passwd

  Exemple :

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
```

- Dans l'ordre : Login, mot de passe (x si crypté), UID, GID (du groupe principal), commentaire (nom de l'utilisateur), répertoire home, shell.

4.2) LISTER LES UTILISATEURS ET LES GROUPES

- Pour lister les logins des utilisateurs uniquement :
  - Soit on filtre le contenu du fichier /etc/passwd :
    cut -d: -f1 /etc/passwd
  - Soit on utilise la commande « compgen » si elle est disponible :
    compgen -u

4.2) LISTER LES UTILISATEURS ET LES GROUPES

- L'ensemble des groupes est listé dans le fichier /etc/group

  Exemple :

  ```
  root:x:0:root
  daemon:x:1:daemon
  ```

- Dans l'ordre : nom du groupe, mot de passe du groupe, GID, utilisateurs du groupe (séparés par une virgule)

4.2) LISTER LES UTILISATEURS ET LES GROUPES

- Pour lister les noms des groupes uniquement :
  - Soit on filtre le contenu du fichier /etc/groups :
    cut -d: -f1 /etc/groups
  - Soit on utilise la commande « compgen » si elle est disponible :
    compgen -g

4.2) LISTER LES UTILISATEURS ET LES GROUPES

- « id »

  Affiche l'UID, le GID et la liste des groupes d'un utilisateur

  - Syntaxe de base :

    id [user]

  - Paramètre :

    - user : Nom de l'utilisateur dont on souhaite afficher les informations. Si non-spécifié, affiche les informations de l'utilisateur exécutant la commande.
      Il est également possible de spécifier plusieurs utilisateurs (séparés par des espaces).

4.2) LISTER LES UTILISATEURS ET LES GROUPES

- « groups »

  Affiche la liste des groupes d'un utilisateur

  - Syntaxe de base :

    groups [user]

  - Paramètre :

    - user : Nom de l'utilisateur dont on souhaite afficher les informations. Si non-spécifié, affiche les informations de l'utilisateur exécutant la commande.
      Il est également possible de spécifier plusieurs utilisateurs (séparés par des espaces).

4.3) Créer, modifier et supprimer des utilisateurs et des groupes

4.3) CRÉER, MODIFIER ET SUPPRIMER DES UTILISATEURS ET DES GROUPES

- « adduser »

  Crée un utilisateur de façon interactive

  - Syntaxe de base :

    adduser *[options] utilisateur [groupe]*

  - options (liste partielle) :
    - -disabled-login : Empêche l'utilisateur de se connecter.
    - -home : Permet de fixer l'emplacement du répertoire /home
    - -no-create-home : Ne crée pas de répertoire home

4.3) CRÉER, MODIFIER ET SUPPRIMER DES UTILISATEURS ET DES GROUPES

- utilisateur : Nom de l'utilisateur à créer (minuscules et chiffres uniquement)
- groupe : Nom du groupe dans lequel sera placé l'utilisateur (Le groupe DOIT exister)

Notes :

- « adduser » demandera interactivement le mot de passe à créer pour l'utilisateur.
- Si aucun groupe n'est spécifié, un groupe du même nom que l'utilisateur sera créé.

4.3) CRÉER, MODIFIER ET SUPPRIMER DES UTILISATEURS ET DES GROUPES

- « addgroup »

  Crée un groupe d'utilisateurs de façon interactive

  - Syntaxe de base :

    addgroup *[options] groupe*

  - options : Cf. « man addgroup »

4.3) CRÉER, MODIFIER ET SUPPRIMER DES UTILISATEURS ET DES GROUPES

- « useradd » et « groupadd »

  Crée un utilisateur / groupe d'utilisateurs de façon non-interactive
- Particulièrement utile pour utiliser dans des scripts.
- Référez-vous à la documentation man pour les différentes options.

4.3) CRÉER, MODIFIER ET SUPPRIMER DES UTILISATEURS ET DES GROUPES

- « deluser »

  Supprime un utilisateur

  - Syntaxe de base :

    deluser *utilisateur*

  - utilisateur : Utilisateur à supprimer

IMPORTANT !!! OPERATION IRREVERSIBLE !

4.3) CRÉER, MODIFIER ET SUPPRIMER DES UTILISATEURS ET DES GROUPES

- « groupdel »
  Supprime un groupe
  - Syntaxe de base :
    groupdel *groupe*
  - groupe : Groupe à supprimer
- Important : Vous ne pouvez pas supprimer le groupe primaire d'un utilisateur. Chaque utilisateur utilisant ce groupe doit d'abord être supprimé (ou son groupe primaire modifié). De même, vérifiez qu'aucun fichier n'appartient à ce groupe avant de le supprimer ! Enfin, comme pour deluser, l'opération est **irréversible** !

4.3) CRÉER, MODIFIER ET SUPPRIMER DES UTILISATEURS ET DES GROUPES

- « passwd »

  Modifie le mot de passe d'un utilisateur

  - Syntaxe de base :

    passwd *[utilisateur]*

  - utilisateur : Utilisateur dont il faut changer le mot de passe. Si non spécifié, modifie le mot de passe de l'utilisateur courant

- « usermod » : Permet de modifier un utilisateur. Cf. « man ».

4.3) CRÉER, MODIFIER ET SUPPRIMER DES UTILISATEURS ET DES GROUPES

- Ajouter un utilisateur à un groupe existant :

  adduser *nom_utilisateur nom_groupe*

- Plus de détails sur les opérations d'utilisateurs et de groupes :

  https://doc.ubuntu-fr.org/tutoriel/gestion_utilisateurs_et_groupes_en_ligne_de_commande

5) Gestion des paquets et utilisation à distance

5.1) Le gestionnaire de paquets

5.1) LE GESTIONNAIRE DE PAQUETS

- Rappel : Linux fonctionne avec un ensemble de « paquets »
- Un paquet correspond à un logiciel, ou une partie de logiciel répondant à un besoin bien précis
- Equivalent d'un programme d'installation sous Windows
- Un paquet peut avoir besoin d'autres paquets pour fonctionner : On parle de « dépendances ».

5.1) LE GESTIONNAIRE DE PAQUETS

- Les paquets sont regroupés sur des serveurs appelés « dépôts »
- Chaque distribution propose un ou plusieurs dépôts, contenant un grand nombre de paquets. On peut trouver par exemple un dépôt pour la version stable actuelle de la distribution, un dépôt pour les mises à jour de sécurités, un dépôt pour les versions « bêta » (en cours de test), etc.
- On installe un paquet avec un installeur. La gestion des paquets, leurs dépendances et leurs mises à jour sont effectuées par un gestionnaire.

5.1) LE GESTIONNAIRE DE PAQUETS

- Il existe différents gestionnaires de paquets et installeurs :
  - Le gestionnaire APT et l'installeur DPKG pour les distributions Debian et dérivées
  - DNF & RPM pour Fedora
  - Pacman pour Arch Linux
  - emerge pour Gentoo
  - etc.
- Chaque distribution fonctionne généralement avec un gestionnaire de paquet et un installeur bien précis. Pour Debian, c'est APT & DPKG.

5.2) Installer/mettre à jour des paquets

5.2) INSTALLER/METTRE À JOUR DES PAQUETS

Principe général :

- On commence **toujours** par mettre à jour la liste des paquets dans le gestionnaire.
- Une fois la liste à jour, on peut effectuer les mises à jour, ou installer un nouveau paquet.
- Seule la suppression de paquet ne nécessite pas de mettre à jour la liste des paquets.

5.2) INSTALLER/METTRE À JOUR DES PAQUETS

- « `apt-get` » et « `apt` »

  Ajoute/supprime des paquets (programmes) sous linux

  - Syntaxe de base :

    `apt operation [options] [paquet1 [paquet2 [paquet3 [etc.]]]]`

  - operation :

    - update : Mets à jour la liste des paquets

      RAPPEL : On commence toujours par effectuer un « `apt update` » !

5.2) INSTALLER/METTRE À JOUR DES PAQUETS

- upgrade : Mets à jour les paquets installés. Aucun paquet n'est supprimé. Si un paquet doit être supprimé pour qu'un autre soit mis à jour, la mise à jour de ce dernier ne se fera pas.
- full-upgrade (anciennement dist-upgrade) : idem upgrade, en supprimant les paquets nécessaires pour réaliser la mise à jour dans son ensemble.
- install : Installer le ou les paquets spécifiés en paramètre
- remove : Supprimer le ou les paquets spécifiés en paramètre
- clean : Efface du disque dur les packages téléchargés
- autoremove : Supprime les paquets qui ne sont plus utilisés.

5.2) INSTALLER/METTRE À JOUR DES PAQUETS

- options (liste partielle) :
  - -f : Avec install ou remove, permet de réparer un système dont les dépendances sont défectueuses
  - -s : Fait une simulation des actions à mener sans rien toucher au système.
  - -y : Répond automatiquement oui à toutes les questions (**Fortement déconseillé !**)
  - -purge : Avec remove : supprime tout ce qui peut l'être, y compris les fichiers de configuration
  - -reinstall : Réinstalle les paquets avec leur version la plus récente.

5.2) INSTALLER/METTRE À JOUR DES PAQUETS

- « `apt-cache` »

  Recherche des paquets (programmes) sous linux

  - Syntaxe de base :

    `apt-cache` *`operation [options] [chaine ou nom de paquet]`*

  - operation :

    - search : Recherche les paquets contenant la chaine indiquée.

      Par exemple : « apt-cache search apache » recherche tous les paquets concernant le logiciel apache

5.2) INSTALLER/METTRE À JOUR DES PAQUETS

- show : suivi d'un nom de paquet spécifique, affiche les informations générales concernant ce paquet (Catégorie, développeur, plateforme, description, site web, etc.)
- showpkg : suivi d'un nom de paquet spécifique, affiche les informations techniques du paquet (notamment les dépendances de celui-ci)

- options (liste partielle) :
  - -n : Avec search (avant la chaine recherchée), permet de rechercher uniquement dans le nom du paquet, et non dans la totalité des informations du paquet.

- Plus de détails sur apt-cache à l'adresse suivante : https://doc.ubuntu-fr.org/apt-cache

5.2) INSTALLER/METTRE À JOUR DES PAQUETS

- Procédure type de mise à jour :
  1. On met à jour la liste des paquets : « `apt update` »
  2. On met à jour les paquets, sans suppression : « `apt upgrade` »
  3. Si apt upgrade a indiqué que certains paquets ne seront pas mis à jour au début de son exécution, on lance « `apt full-upgrade` » et on vérifie que les paquets qui seront supprimés ne sont vraiment plus utilisés
  4. Si apt upgrade ou full-upgrade a indiqué que des paquets n'étaient plus utilisés (et que c'est effectivement le cas, bien sûr), on les nettoie avec « `apt autoremove` ».

5.2) INSTALLER/METTRE À JOUR DES PAQUETS

- De façon générale, sur un système ou tous les paquets ont été installés avec un gestionnaire de paquet, il n'y a pas de risque particulier de supprimer un paquet qu'il ne fallait pas, puisque le gestionnaire de paquets s'assure du respect de l'ensemble des dépendances des paquets.
- Il est toutefois possible d'installer manuellement un paquet téléchargé depuis internet, sans passer par un dépôt et un gestionnaire de paquets.
- On utilise alors directement l'installeur.

5.2) INSTALLER/METTRE À JOUR DES PAQUETS

- « dpkg »

  Ajoute/supprime des paquets (programmes) sous linux

  - Syntaxe de base :

    ```
    dpkg [options] paquet
    ```

  - options :
    - -i : installe un paquet
    - -r : supprime un paquet
    - -P : purge le paquet (comme apt avec l'option -purge)

5.2) INSTALLER/METTRE À JOUR DES PAQUETS

- « `dpkg-reconfigure` »

  Reconfigure un paquet déjà installé

  - Syntaxe de base :

    `dpkg-reconfigure` *`[options] paquet`*

- Si le paquet dispose d'écrans de configurations pendant son installation, ils seront réaffichés pour pouvoir changer les réglages.
- Plus d'infos sur les options avec « man »

5.3) Utiliser linux à distance

5.3) UTILISER LINUX À DISTANCE

- La quasi-totalité des distributions Linux proposent par défaut un utilitaire permettant de se connecter à distance sur une machine Linux : SSH
- SSH permet par défaut d'accéder à un terminal sur une machine distante.
- SSH permet aussi, avec la configuration adéquate, d'accéder à l'interface graphique (aussi appelé serveur X) d'une machine Linux
- SSH permet encore d'autres usages plus poussés, qui ne seront pas abordés ici.

5.3) UTILISER LINUX À DISTANCE

- Pour se connecter à une machine, il est nécessaire de connaître son adresse IP.
- On obtient l'adresse IP d'une machine avec la commande « `ip a` ».
- On peut aussi se connecter sur une machine d'un même réseau avec son nom d'hôte (si la configuration réseau le permet). On peut connaître le nom d'hôte d'une machine avec la commande « `hostname` ».

Note : les commandes ip et hostname seront détaillées dans un autre module

5.3) UTILISER LINUX À DISTANCE

- « `ssh` »

  Permet de se connecter à distance à un ordinateur linux

  - Syntaxe de base :

    ```
    ssh [options] user@hostname [commande]
    ```

  - user : login avec lequel se connecter. Si non spécifié, prend le login de la session courante
  - hostname : adresse IP ou nom d'hôte de la machine distante
  - commande : Si spécifié (entre guillemets), exécute la commande sur l'hôte distant au lieu d'ouvrir un shell

5.3) UTILISER LINUX À DISTANCE

- options (liste partielle) :
  - -X : Active le transfert du serveur X (Le X est en Majuscule)
- SSH permet également de se connecter grâce à un couple clef privée/clef public RSA. Plus d'informations sur internet
- Sous Windows, vous pouvez faire du SSH grâce au logiciel libre et gratuit « Putty », ou avec le WSL. Vous pouvez également transférer le serveur X avec Putty en installant Xming, ou en utilisant le WSL.

5.3) UTILISER LINUX À DISTANCE

- Important : Lorsqu'on se connecte en SSH, l'utilisateur indiqué lors de la connexion doit correspondre à un compte utilisateur de la machine **distante**, et est totalement indépendant du compte que l'on utilise sur la machine source.
- Les droits obtenus une fois connecté sont ceux de ce même compte utilisateur, indépendamment des droits que l'on a sur la machine source !

5.3) UTILISER LINUX À DISTANCE

- « `scp` »

  Permet de copier des fichiers sur un hôte distant de façon sécurisée via SSH

  - Syntaxe de base :

    ```
    scp [options] source destination
    ```

  - source et destination : soit un répertoire local, soit une machine distante, sous la forme *user@hostname:chemin ou « chemin » est le répertoire distant souhaité* (Absolu ou relatif)

    Exemple : scp monfichier.txt toto@192.168.10.37:~/Documents/

5.3) UTILISER LINUX À DISTANCE

- options :
  - -p : Préserve les informations temporelles (date de modification et accès), ainsi que les modes
  - -r : Copie récursive
- Important : Il n'est **pas possible** de copier un fichier d'une machine distante vers une autre machine distante !
- Pour faire cela, il faut :
  - Se connecter sur la première machine distante en SSH
  - Effectuer un scp depuis la première machine distante (qui est alors locale) vers la seconde.

5.4) Palier aux coupures de réseau en SSH

5.4) PALIER AUX COUPURES DE RÉSEAU EN SSH

- SSH permet de se connecter au travers d'un réseau, qu'il soit local ou que ça soit Internet.
- Il peut arriver que la connexion soit coupée, pour diverses raisons.
- Toute commande étant lancée dans le terminal obtenu par SSH sera donc automatiquement interrompue à la fermeture, pouvant entraîner une perte d'informations ou des bugs, selon ce qui était effectué.

5.4) PALIER AUX COUPURES DE RÉSEAU EN SSH

- Le paquet « screen » permet de palier à ce problème en créant un terminal virtuel, indépendant du terminal dans lequel il a été lancé.
- En cas de coupure réseau, il suffit de se reconnecter en SSH et de « rattacher » son terminal virtuel Screen pour reprendre là où on en était.
- Si Screen n'est pas disponible sur votre machine, vous pouvez l'installer avec le gestionnaire apt :
  - `apt update`
  - `apt install screen`

5.4) PALIER AUX COUPURES DE RÉSEAU EN SSH

- « screen » :

  Permet de lancer un ou plusieurs terminaux virtuels qui persistent après la déconnexion. Particulièrement utile pour de l'administration en SSH.

- Syntaxe :

  screen [options]

- Options :
  - -S *nom_session* : Permet de nommer la session screen avec le nom « nom_session »

5.4) PALIER AUX COUPURES DE RÉSEAU EN SSH

- -r : Permet de rattacher un screen existant.
  Si plusieurs screens sont chargés, la commande affiche la liste des screens possibles et la syntaxe pour appeler le screen voulu
- -x : Idem -r, mais permet en plus de partager le screen avec une autre session (du même utilisateur)

5.4) PALIER AUX COUPURES DE RÉSEAU EN SSH

- Principales commandes de screen : Ctrl + a, on relâche les touches, puis :
  - Ctrl + d : Permet de détacher le screen (pour pouvoir le rattacher ensuite avec -r ou -x)
  - c : Crée un nouveau terminal
  - " : Affiche la liste des terminaux du screen et permet de choisir lequel utiliser (avec les flèches)
  - n / p : Passe au terminal suivant / précédent
  - A : Permet de changer le nom du terminal, pour les repérer plus facilement

5.4) PALIER AUX COUPURES DE RÉSEAU EN SSH

- Fermer un terminal virtuel au sein d'une session screen :
  Ctrl + d
- A noter que ce raccourci fonctionne pour n'importe quel terminal, y compris en dehors de screen.
- Si vous fermez le dernier terminal virtuel d'une session screen, cela termine également la session screen.
- Plus d'infos sur : https://doc.ubuntu-fr.org/screen

6) Programmation shell

6.1) Introduction aux scripts shell

6.1) INTRODUCTION AUX SCRIPTS SHELL

- Un script shell porte généralement l'extension .sh
- Doit avoir les droits d'exécution pour fonctionner de manière autonome
- Plusieurs shells existants :
  - sh : Bourne Shell. L'ancêtre de tous les shells.
  - bash : Bourne Again Shell. « sh amélioré », disponible par défaut sous Linux et Mac OS X.
  - ksh : Korn Shell. Un shell puissant assez présent sur les Unix propriétaires, mais aussi disponible en version libre, compatible avec bash.
  - zsh : Z Shell. Shell assez récent reprenant les meilleures idées de bash, ksh et tcsh.

6.1) INTRODUCTION AUX SCRIPTS SHELL

- La 1ère ligne d'un script shell indique le shell utilisé. On l'appelle « shebang » :
  `#!/bin/bash`
- Le shell doit bien sûr être installé sur la machine exécutant le script.
- Nous utiliserons **bash**, car très répandu et assez riche en fonctionnalités
- La suite est un enchainement de commandes de votre choix
- Une ligne commençant par # est un commentaire

6.1) INTRODUCTION AUX SCRIPTS SHELL

Lancer un script shell : 2 méthodes

- Directement :

  `./monscript.sh`

  - **Le script doit avoir les droits d'exécution !**
  - La 1ère ligne doit bien indiquer le shell à utiliser !

- En lançant le shell et en lui donnant le script en paramètre :

  `/bin/bash monscript.sh`

  ➔ Pas besoin des droits d'exécution, le droit de lecture suffit.

6.2) Les variables en shell

6.2) LES VARIABLES EN SHELL

- Déclarer une variable :
  `mavariable='mavaleur'` (Pour un numérique, pas de ')
- **Attention ! Aucun espace autour du = !**
- Pour mettre une ' dans la chaine, il faut l'échapper avec \ (cf. diapo 17)
- Exemple :

```
mavariable='Le cours d\'aujourd\'hui porte sur les scripts shell'
```

6.2) LES VARIABLES EN SHELL

- Afficher une variable :
  `echo $mavariable`
- Attention ! On met un $ pour faire appel au contenu de la variable, mais pas pour la déclarer/l'affecter (on manipule alors le contenant) !
- `echo` peut également afficher toute chaine de caractères
- `echo` dispose de plusieurs options. Pour plus d'infos : `man echo`

6.2) LES VARIABLES EN SHELL

- Une variable `$toto` peut aussi s'écrire `${toto}`
- Cela permet de « noyer » une variable dans une chaine de caractère et de mieux l'identifier.
- Exemple : J'ai une variable « taille » qui vaut « 4 ».

  `echo "Taille du disque : $tailleTo":`

  Bash ne saura pas quel est le nom de la variable.

  `echo "Taille du disque : ${taille}To"` : Plus de confusion !

6.2) LES VARIABLES EN SHELL

Les quotes :

Il existe trois types de quote en shell :

- Les simples quotes : '' : Le contenu n'est pas analysé et est traité de façon brute : `echo 'je m\'appelle ${prenom}.'` affiche « Je m'appelle ${prenom}. », littéralement.
- Les doubles quotes : " " : Le contenu est analysé et traité. Par exemple, si vous mettez une variable dans la chaine entre "", le contenu de la variable sera utilisé et non le nom variable lui-même. Si ${prenom} vaut « Julien » :
  `echo "je m'appelle ${prenom}."` affiche « Je m'appelle Julien »

6.2) LES VARIABLES EN SHELL

- Les back quotes : ` ` : Ce qui se trouve entre `` est exécuté par bash et c'est le résultat de l'exécution qui est utilisé dans le traitement
- Alternative aux back quotes : $()
- `useradd --password `mkpasswd ${monmotdepasse}` $login`
  est équivalent à :
  `useradd --password $(mkpasswd ${monmotdepasse}) $login`
- La commande « `mkpasswd ${monmotdepasse}` » va afficher sur la sortie standard la version cryptée du mot de passe contenu dans `${monmotdepasse}`.

6.2) LES VARIABLES EN SHELL

- Les `` ou les $() vont récupérer la sortie standard de la commande « `mkpasswd` » et l'insérer dans la ligne de commande. Ainsi, c'est comme si on avait (pour le mot de passe « toto ») :

```
useradd --password
$y$j9T$eddkkiqAPrYw/R.romm7b1$xQzPEZC4tVf.P0ik3mnCwQ1eyRYQ5a2Ns3euR2Gv8t6
$login
```

- Avantage de $() : Il est imbriquable. Par exemple :

```
useradd --password $(mkpasswd $(echo ${monmotdepasse})) $login
```

6.2) LES VARIABLES EN SHELL

- $() et échappement de guillemets : Pas d'échappement dans le $() ! Cela est du à l'ordre d'exécution des commandes par bash. Exemple :

```
echo "\"Toto\" en crypté donne : $(mkpasswd "Toto")"
```

- Bash exécute d'abord le contenu du $(), remplace le $() par la sortie standard de la commande exécutée dans le $(), puis seulement à ce moment-là, il exécute la commande principale. Ce qui donne, après exécution du mkpasswd :

```
echo "\"Toto\" en crypté donne : $y$j9T$LWvtsUY5i.B9Up2Z6HM950$aKtAUmErHosB7Z0PVi0tdBvoHmGN1ycuACT7LKIWcr2"
```

- ➔ Il n'y a plus de guillemets qui font conflit au moment de l'exécution.

6.2) LES VARIABLES EN SHELL

- Demander à l'utilisateur de saisir une valeur :

```
read
```

- Exemple :

```
read nom
echo "Bonjour ${nom} !"
```

Demande, puis salue l'utilisateur par son nom

6.2) LES VARIABLES EN SHELL

- Saisir plusieurs variables d'affilée :

```
read nom prenom
echo "Bonjour ${prenom} ${nom} !"
```

- `read` lit mot par mot (un mot est séparé d'un autre par un espace). Chaque mot est affecté aux variables spécifiées dans l'ordre où elles sont données.
- La dernière variable récupère tous les mots restants s'il y en a plus que le nombre de variables données.

6.2) LES VARIABLES EN SHELL

- -p : Affiche en plus un message pour l'utilisateur.
- Exemple :

```
read -p 'Entrez votre nom : ' nom
echo "Bonjour ${nom} !"
```

- -s : Masque le texte saisi (pratique pour un mot de passe, par exemple)

6.2) LES VARIABLES EN SHELL

- -t s : Renvoie une valeur vide dans la variable au bout de s secondes.
- Exemple :

  read -t 15 -p 'Entrez votre nom dans les 15 secondes qui suivent : ' nom

- **Important ! La doc de read se trouve dans le man de bash !**

6.3) Calculs mathématiques

6.3) CALCULS MATHÉMATIQUES

- Bash ne peut pas faire directement de calculs mathématiques
- La commande « let » permet d'effectuer des calculs et de récupérer le résultat dans une variable. Les calculs se font uniquement avec des entiers.
- Opérations prises en charge :
  - Addition : +
  - Soustraction : -
  - Multiplication : *
  - Division : /
  - Puissance : **
  - Modulo : %

6.3) CALCULS MATHÉMATIQUES

- let peut également utiliser des variables (Attention : sans le $!)
- Quelques exemples :

```
let "a = 5 * 3"  # ${a} = 15
let "a = 4 ** 2" # ${a} = 16
let "a = 8 / 2"  # ${a} = 4
let "a = 10 / 3" # ${a} = 3
```

```
let "a = 10 % 3" # ${a} = 1
let "a = 5"      # ${a} = 5
let "b = 2"      # ${b} = 2
let "c = a + b"  # ${c} = 7
```

Calculs en nombres entiers !

6.3) CALCULS MATHÉMATIQUES

- A l'instar de nombreux langages de programmation, il est possible de contracter les opérations :

```
let "a = a * 3"
```

est équivalent à :

```
let "a *= 3"
```

- Notez que dans l'expression de calcul, il est possible de mettre des espaces.

6.3) CALCULS MATHÉMATIQUES

- Pour faire des calculs de temps (exprimés en secondes) :
  - Récupérez la date du moment exprimée en timestamp (cf. diapos 119 & 120):
    ```
    dateDebut=$(date +%s)
    ```
  - Récupérez cette même date à un autre moment de votre script :
    ```
    nouvelleDate=$(date +%s)
    ```
  - Faites la soustraction et vous obtenez le nombre de secondes écoulées :
    ```
    let "tempsEcoule = nouvelleDate - dateDebut"
    echo "Le script a déjà duré ${tempsEcoule} seconde(s)."
    ```

6.3) CALCULS MATHÉMATIQUES

- Autres possibilités pour effectuer des calculs : `$(( ))` et `bc`.
- Cf. https://fr.wikibooks.org/wiki/Programmation_Bash/Calculs

6.4) Paramètres et codes de retour

6.4) PARAMÈTRES ET CODES DE RETOUR

- On peut indiquer des paramètres au script :
  ./monscript.sh valeur1 valeur2 etc...
- Chaque paramètre est enregistré dans des variables numérotées :
- `$0` = monscript.sh (le nom du script, en fait)
- `$1` = valeur1
- `$2` = valeur2, etc.
- Ces variables sont automatiquement renseignées par Bash.

6.4) PARAMÈTRES ET CODES DE RETOUR

- `$#` contient le nombre de paramètres envoyés (sans compter `$0`)
- Si on veut plus de 9 paramètres : « shift » (Ou $10, $11, etc. sur Bash >= v3)
- shift décale d'un cran vers la gauche les paramètres :

```
./monscript.sh val1 val2
echo "Le paramètre 1 est $1"              # $1 = val1
shift
echo "Le param. 1 est maintenant $1"      # $1 = val2
```

6.4) PARAMÈTRES ET CODES DE RETOUR

- En bash, toutes les commandes renvoient un code de retour, dont la valeur peut aller de 0 à 255.
- Par convention, 0 signifie que la commande s'est bien exécutée. Les valeurs supérieures à 0 représentent différents cas d'erreurs. Ceux-ci sont propres à chaque commande, et sont documentés dans le « man » de la commande concernée.

6.4) PARAMÈTRES ET CODES DE RETOUR

- On peut récupérer le code de retour d'une commande dans la variable « `$?` », juste après l'exécution de cette commande.
- Ainsi, si j'exécute le script suivant :

```
ls toto.txt
echo "Code de retour : $?"
```

Le code de retour vaudra « 0 » si la commande ls s'est bien exécutée, et une valeur plus élevée si il y a eu un problème (par exemple, si le fichier n'existe pas).

6.4) PARAMÈTRES ET CODES DE RETOUR

- Important !

  $? est automatiquement rempli après chaque commande ! Si vous souhaitez conserver un résultat en particulier, pensez à le copier dans une variable !

- Exemple :

```
grep -q "toto" monfichier.txt # (cf. diapos 82 & 83 )
retourgrep=$?
echo "toto trouvé dans monfichier.txt (0=vrai) : ${retourgrep}"
echo "Code de retour de echo : $?"
```

6.4) PARAMÈTRES ET CODES DE RETOUR

- On peut également renvoyer un code de retour dans un script grâce aux mots réservés « `return` » ou « `exit` »

  Exemple :

  ```
  return 0
  ```

- Si rien n'est indiqué dans un script, il renverra automatiquement le code « 0 » à la fin de son exécution.

6.5) Tests et conditions

6.5) TESTS ET CONDITIONS

- Un test en bash se fait avec la commande « `test` ».
  Par exemple :
  ```
  test "${login}" = "toto"
  ```
- Cette commande peut également s'écrire « [ ] »
  Par exemple :
  ```
  [ "${login}" = "toto" ]
  ```
- La commande renvoie 0 si le test est vrai, 1 si le test est faux, 2 ou plus en cas d'erreur.

6.5) TESTS ET CONDITIONS

- Pour exploiter un test et en faire une condition « SI », on utilise la syntaxe suivante :

```
if [ test ]                    # Attention aux espaces à
then                           # l'intérieur des [ ] !
    echo "C'est vrai !"
fi
```

- Le « if » vaudra « vrai » si le test renvoie 0, et faux si le test renvoie une autre valeur. A noter que cela fonctionne avec n'importe quelle commande !

6.5) TESTS ET CONDITIONS

- Condition SI – SINON :

```
if [ test ]
then
    echo "C'est vrai !"
else
    echo "C'est faux :( "
fi
```

6.5) TESTS ET CONDITIONS

- Condition SI – SINON SI – SINON :

```
if [ test ]
then
    echo "C'est vrai !"
elif [ test2 ]                  # Autant de elif qu'on veut.
then                            # N'oubliez pas le "then"
    echo "Ok pour test 2" # après le "elif" !
else                            # Un seul else par if,
    echo "KO"                   # et en dernier !
fi
```

6.5) TESTS ET CONDITIONS

- 3 types de tests possibles :
  - Test sur chaine de caractère
  - Test sur nombre
  - Test sur fichier
- Tests sur chaine de caractère : Avec variable ou chaine bash :

```
if [ "${nom}" = "prenom" ]    # Notez les espaces
then […]                      # autour du = !
```

6.5) TESTS ET CONDITIONS

- Opérateurs de comparaison de chaine :
- `"${chaine1}" = "${chaine2}"` : Vrai si les chaines sont égales (sensible à la casse)
  Possibilité d'écrire aussi == à l'instar de Java, C, PHP, etc.
- `"${chaine1}" != "${chaine2}"` : Vrai si les chaines sont différentes.
- `-z "${chaine}"` : Vrai si la chaine est vide
- `-n "${chaine}"` : Vrai si la chaine n'est pas vide
- **IMPORTANT : « = » ne permet PAS de comparer des nombres** !

6.5) TESTS ET CONDITIONS

- Tests sur les nombres
- `${num1} -eq ${num2}` : Vrai si les nombres sont égaux (Ne pas confondre avec =)
- `${num1} -ne ${num2}` : Vrai si les nombres sont différents
- `${num1} -lt ${num2}` : Vrai si ${num1} < ${num2}
- `${num1} -le ${num2}` : Vrai si ${num1} <= ${num2}
- `${num1} -gt ${num2}` : Vrai si ${num1} > ${num2}
- `${num1} -ge ${num2}` : Vrai si ${num1} >= ${num2}

6.5) TESTS ET CONDITIONS

- Tests sur les fichiers
- `-e "${nomfichier}"` : Le fichier existe
- `-d "${nomfichier}"` : C'est un répertoire
- `-f "${nomfichier}"` : C'est un fichier régulier
- `-L "${nomfichier}"` : C'est un lien symbolique
- `-r "${fichier}"` : Le fichier est lisible (r)
- `-w ${fichier}` : Fichier modifiable (w)
- `-x ${fichier}` : Fichier exécutable (x)
- `${fichier1} -nt ${fichier2}` : Fichier 1 plus récent que fichier 2
- `${fichier1} -ot ${fichier2}` : Fichier 1 plus ancien que fichier 2

6.5) TESTS ET CONDITIONS

- Combiner les tests (s'écrit en dehors des [ ]) :

  « Et » logique : `&&`

  « Ou » logique : `||`

Exemple :

```
if [ $# -ge 1 ] && [ "$1" = "koala" ]
then
        echo "Trouvé !"
fi
```

6.5) TESTS ET CONDITIONS

- Inverser un test : « ! »
- Exemple :

```
if [ ! -e "${fichier}" ]
then
    echo "Le fichier n'existe pas"
fi
```

6.5) TESTS ET CONDITIONS

- Tout comme les tests, vous pouvez combiner des commandes en fonction de leur code de retour.
- Exemple :

```
mkdir toto && cp /tmp/monfichier ./toto
```

➔ Le « `cp` » ne s'exécutera que si le « `mkdir` » s'est bien déroulé.

6.5) TESTS ET CONDITIONS

- Combiner les tests (autre notation, directement à l'intérieur des [ ]) :

  « Et » logique : -a

  « Ou » logique : -o

Exemple :

```
if [ $# -ge 1 -a "$1" = "koala" ]
then
        echo "Trouvé !"
fi
```

6.5) TESTS ET CONDITIONS

- Autre exemple :

```
if [ "${fruit}" = "pomme" -o "${fruit}" = "poire" ]
then
    echo "C'est un fruit commençant par \"po\" ! "
fi
```

6.5) TESTS ET CONDITIONS

- Combinaison avancée avec { } (remplace les parenthèses) :

```
if  [ "true" ] || { [ -e /does/not/exist ]  && [ -e /does/not/exist ] ;} ; then echo true; else echo false; fi
```

- Il est possible d'utiliser les parenthèses si on les échappe ou si on les met entre ' ', mais la lisibilité est moindre :

```
if [ \( "true" -o -e /does/not/exist \) -a -e /does/not/exist ]; then  echo true; else echo false; fi
```

ou

```
if [ '(' "true" -o -e /does/not/exist ')' -a -e /does/not/exist ]; then  echo true; else echo false; fi
```

6.5) TESTS ET CONDITIONS

Quelques remarques et rappels :

- [ ] est une **commande** qui renvoie 0 (vrai), 1 (faux) ou 2 ou plus (erreur)
- Il n'est pas possible d'utiliser directement des parenthèses à l'intérieur d'un test [ ]
- Les tests [ ] sont compatibles avec pratiquement toutes les versions de Bash, et également sh
- Il est possible de faire des tests avec l'opérateur [[ ]] à partir de la version 2.02 de bash
- [[ ]] apporte quelques avantages : Opérateur « =~ » disponible (Comparaison avec expression régulière), pas besoin de protéger les variables avec des "", etc.

6.5) TESTS ET CONDITIONS

- [[ ]] n'est PAS une commande, mais un mot-clef de Bash.
- Il est possible d'utiliser le résultat d'une commande plutôt que [ ] ou [[ ]] pour effectuer un test. Le résultat de la commande déterminera le vrai (0) ou le faux (autre valeur)
- Pour un usage plus poussé des tests : https://abs.traduc.org/abs-fr/ch07.html.
- Discussion sur les différences entre [ ] et [[ ]] : https://forum.ubuntu-fr.org/viewtopic.php?id=398332

6.5) TESTS ET CONDITIONS

- Cas pratique : Vérifier si une variable est un nombre :

```
[ ${mavariable} -eq 1 ] 2> /dev/null
if [ $? -eq 0 -o $? -eq 1 ]
then
    echo "C'est un nombre."
else
    echo "Ce n'est pas un nombre."
fi
```

6.5) TESTS ET CONDITIONS

Explications :

- `[ ${mavariable} -eq 1 ] 2> /dev/null`

  ➔ On compare la valeur de la variable au nombre « 1 ». Le code de retour du test est donc un des trois cas suivants :

  - 0 : La valeur de la variable est égale à 1, donc elle vaut 1.
  - 1 : La valeur de la variable est différente de 1, mais la comparaison s'est bien réalisée. La valeur de la variable est donc un nombre.
  - 2 ou + : La comparaison a échoué. La valeur de la variable n'est donc pas un nombre.

6.5) TESTS ET CONDITIONS

- `if [ $? -eq 0 -o $? -eq 1 ]`
  ➔ On regarde le code de retour du test précédent. S'il vaut 0 ou 1, on a bien un nombre entier. S'il vaut 2, ce n'est pas un nombre.
- Attention ! Il faut impérativement utiliser le « `-o` » et non « `||` ». En effet, avec « `||` », vous effectuez deux tests différents :
  `if [ $? -eq 0 ] || [ $? -eq 1 ]` ➔ Ne fonctionne pas
  Explication : Au premier test, $? contient le code de retour du test de la variable vu à la diapo précédente, mais au 2nd test, $? contient le code de retour du 1er test du « if » actuel. On ne testerait donc pas le bon code retour.

6.5) TESTS ET CONDITIONS

- Enchainer plusieurs tests sur la même variable : « case »

```
case $1 in
    "Bruno")                        # Premier cas de figure : se termine par )
        echo "Salut Bruno !"        # Tout le code s'exécute jusqu'au prochain ;;
        ;;                          # Fin du code du cas de figure
    "Michel")                       # On peut aussi utiliser les jokers. Par exemple :
        echo "Bonjour Michel"       # "M*") correspondra à tous les noms commençants
        ;;                          #  par M
    *)                              # *) correspond au cas par défaut, si aucun autre
        echo "Qui es-tu ?"          # test ne valide la valeur de la variable.
        ;;                          # On peut combiner plusieurs valeurs :
esac                                # "Valeur1" | "Valeur2" | "Valeur3")
```

6.6) Les boucles

6.6) LES BOUCLES

- Boucle « Tant que » :

```
while [ test ]
do
        echo "Action en boucle"
done
```

- Les tests sont les même que pour if.

6.6) LES BOUCLES

- Exemple :

```
#!/bin/bash

while [ -z "${reponse}" ] || [ "${reponse}" != 'oui' ]
do
        read -p 'Dites oui : ' reponse
done
```

6.6) LES BOUCLES

- Boucle « for »
- Permet de parcourir une liste de valeurs : Différent de java ou PHP !
- Exemple :

```
for variable in 'valeur1' 'valeur2' 'valeur3'
do
        echo "La variable vaut ${variable}"
done
```

6.6) LES BOUCLES

- Particulièrement intéressant pour parcourir le résultat d'une commande :

```
liste_fichiers=`ls`      # ou : liste_fichiers=$(ls)

for fichier in ${liste_fichiers}
do
        echo "Fichier trouvé : ${fichier}"
done
```

6.6) LES BOUCLES

- Exemple pour renommer les fichiers d'un répertoire :

```
for fichier in `ls`
do
        mv ${fichier} ${fichier}-old
done
```

6.6) LES BOUCLES

- Simuler un for "classique" : On utilise la commande « seq » :

```
for i in `seq 1 10`
do
    echo "n°${i}"
done
```

- « seq » génère tous les nombres allant du 1er au 2e paramètre.

6.6) LES BOUCLES

- Simuler un for "classique" : Autre syntaxe à partir de la version 3 de bash :

```
for i in {1..15}
do
    echo "n°${i}"
done
```

- {x..y} génère tous les nombres allant de x à y, directement par bash et non en passant par une commande.

6.7) Les fonctions

6.7) LES FONCTIONS

- Déclarer une fonction :

```
maFonction()
{
   instructions
}
```

- Une fonction doit toujours être déclarée **avant** son premier appel
- On ne spécifie pas de paramètre entre les parenthèses, même si elle en reçoit.

6.7) LES FONCTIONS

- Appeler une fonction :
  `maFonction param_1 param_2 ... param_n`
- Les paramètres sont facultatifs.
- Dans le code de la fonction, on peut connaître le nombre de paramètres transmis avec `$#`, comme pour les scripts.
- Les paramètres sont accessibles via $0, $1, $2, etc., comme pour un script. Et de même, $0 correspond au nom de la fonction. Le 1er paramètre est donc dans $1.

6.7) LES FONCTIONS

- Comme pour les scripts, on peut renvoyer une valeur numérique depuis une fonction :

```
return 0
```

- Ou encore :

```
exit 0
```

- Si aucun nombre n'est spécifié, return et exit renvoient 0 (encore une fois, comme pour les scripts).

6.7) LES FONCTIONS

- Exemple :

```
isFruitRouge()
{
    if [ "$1" == "fraise" -o "$1" == "framboise" -o "$1" ==
"groseille" ]
    then
        return 0
    else
        return 1
    fi
}
```

← Exécution correcte (ou vrai, en l'occurence) : Code 0

← Exécution incorrecte (ou faux, en l'occurence) : Code 1

6.7) LES FONCTIONS

```
read -p "Saisissez le nom d'un fruit" fruit

if isFruitRouge ${fruit}
then
    echo "Un(e) ${fruit} est un fruit rouge"
else
    echo "Un(e) ${fruit} n'est pas un fruit rouge"
fi
```

« if » sur le code de retour de la fonction.

- Conclusion : la gestion d'une fonction est similaire à un script.