Support du cours
Utilisation des **Systèmes** Informatiques
-
Commandes UNIX/Linux, outils, initiation au scripting en Bash

Pierre Pompidor
pompidor@lirmm.fr
Les supports de cours sont sur le Moodle de la FDS de l'UM
Annales d'examens sur : http://www.lirmm.fr/~pompidor (le caractère mystérieux avant pompidor est un tilde)

Un système d'exploitation consiste en un **ensemble de programmes** (en fait des dizaines de milliers de logiciels, commandes, bibliothèques, démons, pilotes, ...) qui hantent votre machine ; ces programmes permettant d'en développer d'autres, d'utiliser d'autres logiciels, communiquer (client de messagerie, navigateur...) ou jouer... Cet ensemble est plus ou moins riche suivant ce qui est offert (ou chèrement acquis ?) sur le système qui peut être Unix (et son avatar Linux), Windows, MacOs....

Ce polycopié et le cours associé ne prètendent pas du tout couvrir l'ensemble de ces programmes (ce qui dénoterait un détachement définitif d'une vie sociale normale), mais simplement donner quelques informations sur **les commandes (à exécuter dans un terminal)** les plus utiles pour gérer un ordinateur sous système **Linux** de maniére efficace, ainsi qu'à écrire des scripts système qui utilisent ces commandes (via le langage bash).

A la question, "pourquoi Linux ?" - qui est un des "portages" du système Unix sur ordinateurs personnels - je répondrai que c'est le système installé sur la quasi-totalità desmachines des salles de TP du département informatique car de tous les systèmes, il est le plus ouvert et flexible. Il vous permettra de pratiquer tous les langages et quasiment toutes les technologies nécessaires à une vie professionnelle heureuse. Par ailleurs, ce système est gratuit !

La focalisation du cours sur des commandes assez inquiétantes, permettra d'une part de pouvoir les insérer dans des scripts systèmes (des programmes qui vont automatiser des opérations comlplexes), et d'autre part, évitera à terme de perdre beaucoup trop de temps en cliquant frénétiquement.

Voici quelques remarques sur les chapitres du polycopié :

- Chapitre 1 : **Introduction** : présentation générale d'*Unix*, de *Linux* et d'*Ubuntu*
- Chapitre 2 : **Installation et découverte de *Ubuntu***
  Informations relatives à l'installation et la prise en main de la distribution Ubuntu
- Chapitre 3 : **Environnement**
  Présentation générale de *Linux* et de son environnement.
- Chapitre 4 : **Commandes**
  Gestion de l'ordinateur via des commandes à exécuter virilement dans un terminal.
  En cours, nous ne verrons que les commandes les plus importantes (celles qui sont encadrées).
- Chapitre 5 : **Outils**
  Ce chapitre présente les macro-commandes permettant de se connecter sur une machine distante (principalement *ssh*, *ftp (ou ncftp, sftp...)* et *cURL*, les outils de gestion de versions (en fait GIT) et les éditeurs de textes (pour la programmation et pour la bureautique) les plus importants intégrés à Linux.
- Chapitre 6 : **Interpréteur de commandes / langage de programmation bash**

`Dans le polycopié, les parties importantes sont encadrées.`

Et les touches du clavier sur lesquelles appuyer sont entourées par des "<" et ">"; exemple : <*Tab*>.

Contents

1 Introduction : 4
1.1 Panorama des systèmes d'exploitation pour ordinateurs personnels : 4
1.2 Bref historique d'Unix et de Linux et vue générale : 4

2 Installation et découverte d'Ubuntu : 5
2.1 Pratique de Linux/Ubuntu sur votre ordinateur personnel où Windows est installé 5
2.1.1 Connexion à distance sur un serveur de la FDS avec l'application **x2go** 5
2.1.2 Windows Subsystem for Linux (**WSL** / **WSL 2**) 5
2.1.3 Virtualisation de Ubuntu sous Windows grâce à **VirtualBox** 6
2.2 Prise en main d'Ubuntu 6
2.2.1 Connexion et opération système avec les droits de l'administrateur : 6
2.2.2 Paramétrage du réseau : 6
2.2.3 Parcours de l'arborescence des répertoires du système de fichiers : 7
2.3 Installation de logiciels : 7
2.3.1 Installation de paquets (ou paquetages) pour installer des applications déjé compilées : 7
2.3.2 Installation de sources archivées (codes à compiler) : 7

3 Environnement de base : 8
3.1 Ouverture d'un compte, connexion, environnement, vision des utilisateurs, communication : 8
3.1.1 Ouverture d'un compte : 8
3.1.2 Connexion : 8
3.1.3 Manipulation d'un terminal : 8
3.1.4 Syntaxe et documentation sur les commandes : 9
3.1.5 Informations sur votre machine et la version du système Linux installé : 9
3.1.6 Informations sur les connexions réseaux : 10
3.1.7 Vision des utilisateurs : 10
3.2 Gestion des processus : 10
3.3 Gestion des fichiers : 11
3.3.1 Le système de gestion des fichiers : 11
3.3.2 Quelques commandes de base sur les fichiers : 12
3.3.3 Les mécanismes de redirection, de communication par tubes et de détachement : 12

4 Autres commandes à fréquenter : 13
4.1 Relatives aux interpréteurs de commandes : 13
4.2 Relatives aux places mémoires occupées : 14
4.3 Relatives aux fichiers : 14
4.3.1 Recherche / classification de fichiers dans le système de fichiers : 14
4.3.2 Recherche de données dans le contenu de fichiers texte : 14
4.3.3 Visualisations de fichiers avec mise au format : 15
4.3.4 Comparaisons entre fichiers : 15
4.3.5 Transformations de fichiers : 15
4.3.6 Cryptages, compressions et archivages de fichiers : 16
4.3.7 Impressions de fichiers : 16
4.4 Montage/démontage de systèmes de fichiers : 16

5 Session de travail distante, transfert de fichiers d'une/sur une machine distante : 17
5.1 Ouverture d'une session de travail sur une machine distante : 17
5.2 Transferts de fichiers entre machines : 17
5.3 Récupération d'une ressource prise sur internet : 18

6 Outil de gestion de versions GIT: 18
6.1 Création d'une clef de connexion sécurisée sur votre machine locale et installer git : 18
6.2 Création du dépôt Gitlab de la FDS : 18
6.3 Création du dépôt local et synchronisation avec le dépôt distant : 19

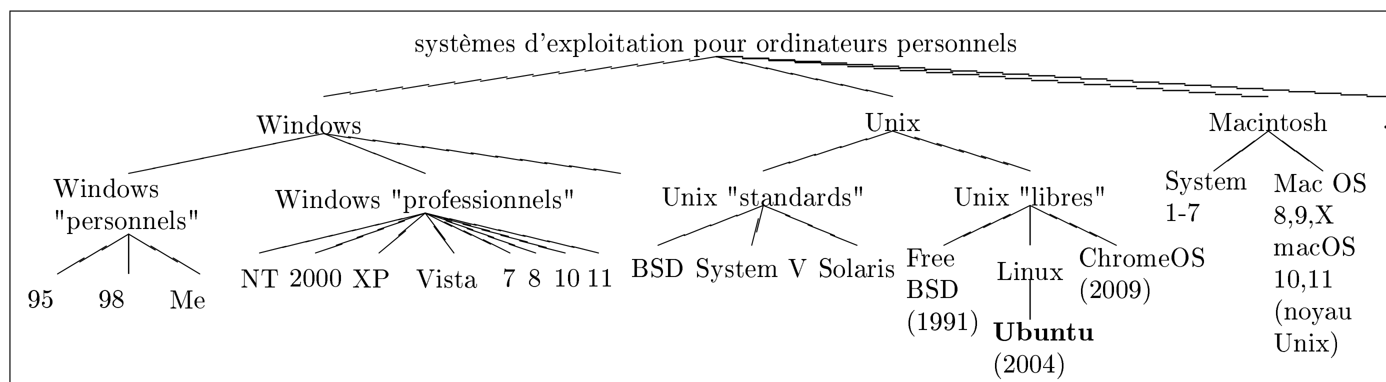
7 Editeurs de texte et environnement de programmation : 19
7.1 Visual Studio Code : 19
7.2 Liste des autres éditeurs "classiques" disponibles : 19
7.2.1 Aperçu de vi : 19
7.2.2 Aperçu d'emacs : 20

8 Introduction au scripting système : 21

9 Scripting bash : 21
9.1 La configuration de bash (et de (t)csh) : 22
9.2 Variables et paramètres : 22
9.2.1 Les variables système : 22
9.2.2 Déclaration / initialisation des variables : 23
9.2.3 Affichage d'une variable : 23
9.2.4 Récupération d'une valeur au clavier : 23
9.2.5 Les tableaux simples (array) : 23
9.2.6 Les tableaux associatifs : 24
9.2.7 Valuation d'une variable par le résultat d'une commande : 24
9.2.8 Calcul arithmétique : 24
9.2.9 Les paramètres de la ligne de commande : 24
9.3 Structures conditionnelles : 24
9.3.1 Structure conditionnelle avec **if** : 24
9.3.2 Utilisation des simples ou doubles crochets : 25
9.3.3 Synthèse des différents emplois de la structure conditionnelle if : 26
9.3.4 Structure conditionnelle avec **case** : 26
9.4 Structures itératives : 26
9.4.1 Structure itérative avec **while** : 26
9.4.2 Structure itérative avec **for** 27
9.5 Lecture de fichiers : 27
9.6 Fonctions : 27
9.7 Pour forcer l'arrét d'un script et récupérer son "statut" : 28
9.8 Quelques exemples de scripts bash : 28
9.8.1 Affichage de tous les répertoires du répertoire courant : 28
9.8.2 Affichage des fichiers réguliers qui contiennent la chaîne passée en paramètre du script : 28
9.8.3 Affichage numérotà deslignes d'un fichier dont le nom est passé en paramètre : 28

1 Introduction :

1.1 Panorama des systèmes d'exploitation pour ordinateurs personnels :



Parts de marché estimées pour les ordinateurs (Wikipedia : Usage share of operating systems) :

- les postes clients : Windows → 61%, MacOS → 31%, **Linux → à peu prés 2%, ...**
- les postes serveurs : **Linux → 37%** et Unix (hors Linux) → 30%, Windows → 33%
- La consécration, les super-calculateurs : **Linux → 97%** et avec les autres systèmes Unix 99%

Parts de marché estimées pour les développeurs professionnels :

- Windows → 45%,
- MacOS → 29%
- **Linux → 25%**

Par ailleurs, il ne faut pas oublier qu'avec **Android**, c'est Linux qui équipe la majorità dessmartphones !

1.2 Bref historique d'Unix et de Linux et vue générale :

UNIX est un système d'exploitation trés fréquemment répandu dans les environnements de recherche, d'enseignement ou de développement. Il n'est lié à aucune architecture ou constructeur particulier. Son interface, longtemps austére, a été améliorée avec le système de gestion de fenêtres X Window (X11, XOrg) et récemment Wayland.
Il a été créé en 1969 au Bell Laboratories par Ken Thompson. Il est rapidement réécrit en C et différentes versions, SYSTEM-V(ATT et Bell Laboratories), BSD (Berkeley), XENIX (Microsoft) ... voient le jour. Une procédure de normalisation a homogéinisé toutes ces versions (normalisation des services offerts et de leurs accés, portabilité au niveau du code source). A partir de 1988 est effectuée l'intégration dans l'environnement de UNIX du système de gestion de fenêtres X Window (projet Athena du MIT) et notamment de la bibliothéque de widgets MOTIF.

Ce cours sera plus spécialement dédié à **Linux** (créé en 1991 par *Linus Torvalds*), une implémentation libre d'UNIX pour les ordinateurs personnels. Linux est **gratuit**, puissant, stable, interfacé par différents **bureaux** comme **Unity**, **KDE** ou **GNOME** (qui eux-mémes reposent sur le système de gestion de fenêtres XWindow).
Linux peut être installé parallélement à d'autres partitions MS-DOS/Windows ou tout autre système d'exploitation. Différentes distributions de Linux existent, la plus populaire étant **Ubuntu**, mais par exemple Debian est aussi trés répandue.

Comme il le sera expliqué, une "partie" d'Ubuntu est intégré à Windows 10/11 (*WSL*).

Remarque : RedHat avec le format **RPM** et Debian ou Ubuntu avec le format **deb**, ont conéu des archives qui permettent d'effectuer une installation, une désinstallation ou une mise à jour de n'importe quel logiciel trés facilement. Ils vérifient les dépendances nécessaires et les installent directement au bon endroit. Ubuntu intégre un logiciel de téléchargement de logiciels (**Synaptic**) trés convivial.

Par ailleurs Unix est un :

- système **multi-utilisateurs**,
- système **multi-téches** ; il gére :

- avec X11, plusieurs fenêtres représentants autant de terminaux différents,
- des processus en arriére plan,
- une répartition optimisée des ressources de l'ordinateur,

- Son système **de développement** comprend :
  - de puissants interpréteurs de commandes appelés shells,
  - des systèmes de fichiers et de processus hiérarchisés,
  - une vision unique des différents types d'entrées-sorties,
  - une réallocation des entrées-sorties des processus (filtres et redirections),
  - des points d'accés aux services offerts par le noyau dans des langages évolués (appels systèmes)

2 Installation et découverte d'Ubuntu :

→ Prévilégiez les versions stables des mois d'avril et d'octobre (4 et 10)

Ubuntu (mot bantou dont la signification essaie de nous rappeler notre unité), est une distribution de Linux (ou plutét des distributions...) qui a les mérites :

- d'étre (comme les autres distributions) exécutable à partir de Windows (ceci est expliqué ci-après) ;
- d'étre basée sur l'abcienne et solide distribution *Debian* ;
- de posséder de multiples pilotes lui permettant de s'adapter à un nombre considérable de machines ;
- d'offrir un système convivial de téléchargement de logiciels (Synaptic)

2.1 Pratique de Linux/Ubuntu sur votre ordinateur personnel où Windows est installé

Pour utiliser Linux/Ubuntu sur un ordinateur personnel Windows, vous avez (au moins) cinq possibilités :

- vous connecter à distance sur un serveur de la FDS avec l'application **x2go**
- mettre en œuvre un sous-système de Windows (>=10) qui permet d'exécuter certains binaires de la distribution Ubuntu ;
- l'utiliser dans une machine virtuelle (sans qu'il soit réellement installé sur votre machine) ;
- l'installer à cété de Windows sur votre disque dur (installation en *dual boot*) que je vous déconseille de faire (et que je ne commente plus).
- en bootant sur une clef bootable Linux (déconseillé)

2.1.1 Connexion à distance sur un serveur de la FDS avec l'application x2go

Pour tout savoir, visitez ce lien :
https://moodle.umontpellier.fr/mod/page/view.php?id=126345
(Vous devrez installer un client à partir de **https://wiki.x2go.org/doku.php/download:start**)

2.1.2 Windows Subsystem for Linux (WSL / WSL 2)

Sur les versions 64 bits de Windows 10/11, un sous-sytéme est installable qui permet d'exécuter certains binaires de la distribution Ubuntu de Linux (depuis sa version 16.04).

C'est assez efficace et voici les liens que je vous recommande pour explorer cette solution :
https://docs.microsoft.com/fr-fr/windows/wsl/install
https://doc.ubuntu-fr.org/wsl

Une fois l'application *bash* installée sous Windows :

- sous bash, les systèmes de fichiers Windows sont accessibles à partir de **/mnt** et les comptes utilisateurs à partir de **/mnt/c/Users** ;

- sous Windows, les systèmes de fichiers Linux sont accessibles à partir de :
  pour WSL : `C:\Users\<NomDeLUtilisateur>\AppData\Local\lxss`
  pour WSL 2 : à partir du partage spécial `\\wsl$` (é saisir dans la zone d'accés rapide de l'explorateur de fichiers)

Attention, il est possible que l'antivirus/pare-feu de Windows s'oppose aux installations de paquets, dans ce cas il faut le désactiver temporairement.

2.1.3 Virtualisation de Ubuntu sous Windows grâce à VirtualBox

VirtualBox, développé maintenant par **Oracle**, permet de virtualiser des systèmes d'exploitation ***invités*** sur un système d'exploitation ***héte***. A peu prés toutes les combinaisons sont possibles (par exemple héberger Linux sous du Windows ou le contraire).

Schématiquement, pour héberger Ubuntu sous Windows, vous devez :

- récupérer l'image ISO de la dernière distribution "stable" d'Ubuntu :
  - se connecter à **http://www.ubuntu.com/download/desktop**
  - downloader la dernière LTS (Long Term Support) **ubuntu-22.04-desktop-amd64.iso** *The Jammy Jellyfish*
- installer VirtualBox :
  - se connecter à **http://www.virtualbox.org/**
  - downloader le **binaire** VirtualBox pour la version de Windows intallée
- lancer VirtualBox et créer une **machine virtuelle** → icéne *Nouvelle* :
  - en la pré-configurant pour Ubuntu (pré-configuration un peu mystérieuse)
  - en lui associant :
    - un lecteur CD (a priori c'est fait automatiquement), mais qui s'en sert encore ?
    - un disque dur virtuel (redimmensionnable, 8G de taille max).
- installer le système invité (Ubuntu) sur le système héte (Windows) : → icéne *Démarrer* :
  - browser l'image ISO : → pour le faire apparaître dans la liste associée à "Choix du média d'installation" ;
  - puis rajouter les additions relatives au système Ubuntu
    (notamment pour pouvoir mieux gérer la fenêtre relative à ce système).

Remplacer le mot *virtualisation* par *émulation* serait un peu impropre car il désigne la traduction des instructions d'un micro-processeur vers un autre, mais bon on peut l'employer...

2.2 Prise en main d'Ubuntu

2.2.1 Connexion et opération système avec les droits de l'administrateur :

Vous devez vous identifier avec le login et le mot de passe que vous avez choisis lors de l'installation du système. Vous êtes par défaut un **utilisateur particulier** qui peut avoir momentanément les droits de l'administrateur (appelé sous les autres distributions **root**). Dans un terminal, vous pourrez exécuter une téche nécessitant les droits de l'administrateur de deux maniéres différentes :

- pour lancer une commande non interfacée : **sudo [commande]** (**sudo -s** pour créer une session) ;
- pour lancer une application interfacée : **gksudo [application]** .

Par exemple, à partir d'un terminal (cherchez-le ;-)), visualisez l'état des partitions par :
sudo fdisk -l ou **sudo parted -l**.

2.2.2 Paramétrage du réseau :

Pour configurer la connexion au réseau, vous devez ouvrir la fenêtre de configuration par **système** → **Administration** → **Reseau**. Cela-dit, votre poste peut être géré par un serveur **DHCP**, Ubuntu a se debrouillant alors tout seul... Si vous deviez configurer une adresse IP fixe, outre celle-ci paramètrez également le sous-réseau et la passerelle.

2.2.3 Parcours de l'arborescence des répertoires du système de fichiers :

Selectionnez : **Raccourcis** → **Poste de travail**, pour ouvrir **l'explorateur de fichiers**.
Remontez jusqu'au dossier (repertoire) racine du système de fichiers, et explorez la hierarchie de repertoires.
Dans la barre d'icénes du haut du navigateur (en dessous des menus) :

- **[F9]** permet de desafficher/visualiser les repertoires dans le panneau de gauche
- la **fléche vers le haut** permet de remonter au répertoire pére
- la **fléche vers la gauche** permet de revenir au positionnement précédent
(essayez la **fléche vers la droite** après être revenu en arriére)
- en cliquant avec le bouton droit de la souris sur l'icéne d'un fichier, vous pouvez opérer un certain nombre de manipulations (que je vous laisse découvrir)

Vous pouvez également utiliser le panneau de gauche pour vous positionner directement sur un répertoire.
Par celui-ci, vous pouvez également fermer ou ouvrir un répertoire.

2.3 Installation de logiciels :

2.3.1 Installation de paquets (ou paquetages) pour installer des applications déjé compilées :

Pour installer de nouvelles applications, de nouvelles bibliothéques (comprenant des briques logicielles), plusieurs possibilités vous sont offertes (dans ce qui suit <*paquet*> désigne un nom de paquet...) :

- par l'application **Synaptic** si elle existe ;
- par l'outil **apt** (*Advanced Packaging Tool*), et plus précisément la commande **sudo apt install** <paquet> qui installe un paquet en essayant de résoudre les dépendances (céd en important également les bibliothéques nécessaires à l'installation)
- par la commande **sudo dpkg -i** <paquet.deb> :
pour des paquets de bas niveau (mais sans que les dépendances ne soient résolues).

La commande *apt* rend obsolète la commande *apt-get*.

Le fichier de configuration **/etc/apt/sources.list** répertorie l'ensemble des **serveurs** = **dépôts** connus contenant des paquets. A partir de ces dépôts, *apt* connaét la liste des paquets disponibles : cette liste peut être mise à jour par la commande **sudo apt update**.
La commande **sudo apt upgrade** met à jour tous les paquets.
La commande **sudo do-release-upgrade** permet d'installer une nouvelle version du système.

2.3.2 Installation de sources archivées (codes à compiler) :

Quelquefois l'application qui vous intéresse n'est pas livrée dans un paquet mais dans une archive (généralement au format **.tar** ou **.tgz**) : dans ce cas cette archive contient les codes sources de l'application qui doivent être recompilés.
*Attention : la commande **tar** sera décrite un peu plus loin.*

Voici les étapes à suivre :

- Téléchargement d'une archive **.tar**, **.tar.gz** ou **.tgz**, l'archive est recopiée par défaut dans le répertoire **Téléchargements / Downloads**
- **tar -xzvf nom_de_l_archive** : désarchivage + décompression → voir ci-après la documentation de cette commande
- lire les fichiers README et INSTALL s'ils existent
- **./configure** : création du fichier de gestion de la compilation **Makefile**
- **./make** : compilation (au sens large du terme (compilation + édition de liens))
- **./make check** : test du résultat de la compilation
- **./make install** : déplacement des binaires dans les répertoires systèmes

3 Environnement de base :

Vous disposez sous Linux de différents environnements graphiques (nommés **bureaux**) comme **Unity**, **GNOME** ou **KDE**. Leur fonctionnement intuitif ne nécessite pas d'explication détaillée dans ce polycopié, hormis deux facilités :

- pour lancer une application qui ne serait pas représentée par une icéne, tapez simultanément **Alt** et **F2**, puis dans la zone de texte l'application à lancer (par exemple le navigateur **Firefox** ou l'effrayant éditeur **emacs**).
- pour explorer les répertoires et ouvrir les fichiers, vous pouvez utiliser l'explorateur de répertoires (pour cela cliquez sur l'icéne représentant une petite maison).

Par ailleurs vous pouvez également ouvrir des **terminaux** où vous saisirez des commandes textuelles qui seront interprétées par un **interpréteur de commandes**. Cette faéon de procéder est pour un utilisateur aguerri, la plus rapide, et la plus efficiente, une minorité de commandes étant interfacée sur le bureau.

3.1 Ouverture d'un compte, connexion, environnement, vision des utilisateurs, communication :

3.1.1 Ouverture d'un compte :

Lors de l'ouverture d'un compte permettant l'utilisation des salles informatiques à la FDS, sont attribués :

- un numéro individuel (uid : user identification),
- une chaîne de caractères de **login** qui sera votre identifiant,
- et un mot de passe.

A la Faculté des Sciences de l'UM, l'ouverture d'un compte est subordonnée à l'inscription administrative.

3.1.2 Connexion :

La connexion s'effectue soit sur une machine isolée ou sur un réseau de machines (anciennement gérées par **NIS** : Network Information Service) ou par un système d'annuaire de type **LDAP**.
Votre connexion est validée si votre login est connu et votre mot de passe encrypté similaire à celui mémorisé dans :

- le fichier */etc/passwd* dans le cas d'une connexion sur une machine isolée :
  cette ligne étant composée comme suit :
  `nom : mot de passe crypté : uid : gid : infos libres : répertoire de travail : shell`
  (uid = numéro d'utilisateur, gid = numéro de groupe)
  Le mot de passe crypté est occulté en mode "shadow" céd recopié dans un autre fichier.
- le fichier *passwd* géré par le serveur NIS dans le cas d'une connexion sur un réseau de machines ;
- via un annuaire LDAP.

après votre connexion, vous pouvez être face :

- à une interface graphique (appelée bureau) ;
- à une interface de type console.

Dans le second cas, pour lancer l'interfaéage graphique utilisez la commande **startx** (cela-dit c'est super mauvais signe). La commande **startx** lance le **serveur X** qui s'occupe de l'affichage de toutes les fenêtres graphiques...

Remarque : Vous pouvez vous connecter plusieurs fois en paralléle sur la méme machine en ouvrant jusqu'é six consoles textuelles et une seule graphique :

- consoles textuelles : de **CTRL ALT F1** à **CTRL ALT F6**
- console en mode graphique : **CTRL ALT F7**

3.1.3 Manipulation d'un terminal :

après familiarisation avec l'interfaéage graphique ayant les mémes fonctionnalités que celles des autrez systèmes pour micro-ordinateurs (Windows ou MacOS), lancez un terminal (nous allons tellement aimer utiliser des terminaux) :

- en cliquant sur une icéne représentant un écran d'ordinateur si elle existe

- en sélectionnant un item “terminal” dans un sous-menu “système” dans le menu principal de votre bureau

Lors de la création d’un terminal, les opérations suivantes sont effectuées :

- Création d’un processus exécutant un **interpréteur de commandes** ou **shell** :
  L’interpréteur de commandes est le programme qui va "comprendre" vos commandes.
  Il existe plusieurs implémentations possibles d’interpréteurs de commandes, les deux les plus fréquentes étant **bash** et **csh**. Pour connaétre l’interpréteur de commandes que vous utilisez par défaut, vous effectuerez la commande suivante dans votre terminal : **echo $SHELL**.
  Sa première implémentation ayant été appelée **shell**, l’interpréteur de commandes est également souvent appelé comme cela (par extension le terminal est souvent aussi appelé ainsi ce qui rend les choses encore plus confuses).
- Configuration de ce processus suivant le contenu d’un fichier de configuration nommé
  - **.bashrc** si votre interpréteur de commandes est **bash**
  - **.cshrc** si votre interpréteur de commandes est **csh** ou **tcsh**
- Affichage de la banniére (message du terminal modifiable par l’utilisateur) :
  - voir à **PS1** si votre interpréteur de commandes est **bash**
  - ou à **prompt** si votre interpréteur de commandes est **csh** ou **tcsh**

après lecture d’une ligne, l’interpréteur de commandes l’analyse pour exécuter les commandes qui y sont recélées.
Il existe deux types de commandes :

- les commandes internes : exécutions directes (comme la commande *cd* vue plus loin),
- et les commandes externes : créations de nouveaux processus.

3.1.4 Syntaxe et documentation sur les commandes :

Une commande Unix saisie dans un terminal a différentes syntaxes, notamment :

- **commande -c_1...c_n paramètres** (où c_i est un caractère définissant une option)
- **commande –mot paramètres** (où mot est un mot explicite)

Pour lire la documentation relative à une commande :

apropos	liste toutes les pages du manuel de toutes les commandes comportant le mot clef donné en paramètre exemple : *apropos directory* (directory signifie répertoire en anglais)
man	affiche les pages du manuel de la commande donnée en paramètre exemple : *man mkdir*

Par ailleurs, si vous ne tapez que les premières lettres d’une commande, l’interpréteur de commande va:

- compléter son nom (s’il n’y a pas d’ambiguété) si vous appuyez une fois sur <**Tab**>
- lister toutes les commandes commenéant par ces lettres si vous appuyez deux fois sur <**Tab**>

3.1.5 Informations sur votre machine et la version du système Linux installé :

uname -a	informations sur la machine locale *nom_système nom_machine révision version Nom_modéle_machine*
lsb_release -a	affichage de la version de Linux
hostname	nom de la machine
lscpu	architecture de votre machine
lspci	périphériques connectés à lacarte mére via des bus PCI
date	affichage de la date et de l’heure *(je sais que cette commande n’a rien à faire lé)*

3.1.6 Informations sur les connexions réseaux :

Un ordinateur reéoit les messages qui proviennent du réseau via une ou plus plusieurs **cartes réseau**.
Chaque carte réseau posséde une **adresse** MAC (*Media Access Control*) unique.

Le message réseau est accompagné d'un **port** (une sorte de numéro de boûte à lettres) qui indique quel est le processus qui doit le prendre en charge (un processus est un programme en cours d'exécution).
Un processus (gérant des messages réseau) et s'exécutant en permanence s'appelle un **service**.

ip a	cartes réseaux et adresses IP rattachées : par exemple *eth0* : première carte ethernet (connexion filaire) *wlan0* : première carte wifi (*Wlan = Wireless Local Area Network*)
service –status-all	état des services (à partir de Ubuntu 16.04)
netstat -antup	connexions réseaux

3.1.7 Vision des utilisateurs :

whoami	login de l'utilisateur
id	uid et gid de l'utilisateur
who	affichage des autres utilisateurs de la machine locale et des numéros de lignes (terminaux) login terminal date_de_connexion
finger	méme fonction que *who*, mais permet en plus : de visionner les connectés d'une machine distante :*finger @machine* de demander des informations sur un utilisateur : *finger login@nommachine*
rwho	permet de connaétre les utilisateurs des machines du réseau local *attention : cela ne fonctionne que si un serveur rwho est en activité sur chaque machien du réseau.*
ou **rusers**	(interrogation de chaque machine)
last	permet d'avoir l'historique des dernières connexions sur la machine

3.2 Gestion des processus :

Un **processus** est :

- un programme en train d'être exécuté
  (en fait plus exactement, un programme qui a été chargé en mémoire centrale) ;
- les données manipulées par ce programme ;
- son contexte d'exécution (registre, pile, liens utilisateurs et système d'E/S ...).

Les processus sont ordonnancés par un ordonnanceur.
Le processus **init 1** est parent des processus shells créés par l'utilisateur.
Caractéristiques d'un processus :

- identification (**PID**),
- identification du processus parent (**PPID**),
- propriétaires,
- groupes propriétaires,
- éventuellement terminal d'attachement (**TTY**),
- priorité,
- différents temps ...

ps :	liste des processus appartenant à un ensemble particulier
	sans options : processus attachés au méme terminal (hors processus XWindow)
	- numéro d'identification (**PID**),
	- numéro terminal (**TTY**)
	- infos sur l'exécution (4 lettres : stoppé, arrété, inactif, priorité réduites ...) (**STAT**)
	- temps cumulé d'exécution (**TIME**)
	- nom du fichier correspondant au prog. exécuté par le processus (**COMMAND**)
	-u (f sur Linux) : format long (avec informations sur la taille mémoire, numéro du pére, etc ...)
	-a (e sur Linux) : liste de tous les processus s'exécutant sur la machine
	-x (e sur Linux) : liste des processus XWindow
top :	affichage dynamique (céd se raffraîchissant toutes les secondes) des processus
	cette commande est interfacée avec le gestionnaire de pages **more**
kill :	pour envoyer un signal à un processus
	-9 numéro_processus pour le tuer de maniére certaine
killall :	pour tuer tous les processus d'un certain nom

3.3 Gestion des fichiers :

3.3.1 Le système de gestion des fichiers :

Un fichier Unix est un fichier sur disque ou une ressource du système (device).
Il peut avoir ou non un contenu sur disque et est de différents types :

- fichier régulier
- répertoire
- lien symbolique
- tube de communication
- ressource (terminaux, claviers, imprimantes, disques physiques et logiques)

Chaque fichier correspond à une entrée (i-noeud, i-node ou index node) dans une table contenant l'ensemble de ses attributs :

- type
- propriétaire
- droits d'accés (lecture, écriture, exécution)
- taille (pour les fichiers sur disque)
- nb de liens physiques
- dates de lecture, de modification du fichier et du noeud
- adresse des blocs utilisés sur le disque (pour les fichiers sur disque)
- identification de la ressource associée (pour les fichiers spéciaux)

Les répertoires permettent d'organiser une arborescence de fichiers qui peuvent avoir des références absolues (/usr/toto) ou relatives (../toto), (**.** répertoire courant, **..** répertoire pére).
Un répertoire UNIX n'est jamais vide (.. et .).

Les répertoires usuels peuvent être rédéfinis par l'administrateur, mais quelques conventions sont respectées :

/bin	commandes non internes du shell (de *bash*)
/dev	noms des fichiers spéciaux associés aux périphériques
/etc	fichiers de configurations, scripts de contrôles de certaines applications
/home	répertoires de travail des utilisateurs
/mnt	les répertoires utilisés pour monter temporairement un système de fichiers (clef USB...)
/tmp	fichiers temporaires
/usr	les applications
/usr/bin	applications installées avec Linux
/usr/local/bin	applications spécifiques
/var	données variables (et notamment sites web)

3.3.2 Quelques commandes de base sur les fichiers :

ls : liste du contenu d'un répertoire
- **l** : type, droits, nb_liens, nom_proprio, groupe_proprio, taille, date
- **d** : n'explore pas les répertoires
- **a** : liste les fichiers "systèmes" commenéant par un "."
- **R** : liste récursive des répertoires
- **h** : affichage human-friendly
- **t** : tri chronologique

cat	liste le contenu d'un fichier (cette commande peut aussi servir à créer des fichiers)
more	liste le contenu d'un fichier page par page appuyez sur la barre d'espacement pour passer d'une page à l'autre sur "q" pour quitter
cp	copie un fichier (*cp -R répertoire_source répertoire_destination* copie récursivement un répertoire)
ln	crée un lien (référence sur un fichier) *ln -s nom_fichier nouveau_nom* (création d'un nouvel inode → lien symbolique)
mv	change le nom ou déplace un fichier *mv nom_fichier nouveau_nom_du_ficher* → renommage *mv nom_fichier répertoire* → déplacement *mv nom_fichier répertoirenouveau_nom_du_ficher* → déplacement et renommage
rm	supprime un fichier *rm -R répertoire* détruit récursivement un répertoire (dangereux)
pwd	donne la réference absolue du répertoire courant
cd	changement de répertoire
mkdir	crée un répertoire
rmdir	détruit un répertoire
chmod	change les droits d'un fichier (mode symbolique ou numérique) **r** pour read, **w** pour write, **x** pour execute, **u** pour user, **g** pour group, **o** pour others, **a** pour all : *chmod go+rx fichier* **4** pour read, **2** pour write, **1** pour execute : *chmod 751 fichier* → droits : rwxr-x–x

umask	note les droits par défaut d'un fichier (*umask 022* pour *rwxr-xr-x*)
chown	change le propriétaire d'un fichier (réservé au superutilisateur)
chgrp	change le groupe d'appartenance d'un fichier

3.3.3 Les mécanismes de redirection, de communication par tubes et de détachement :

Les applications développées doivent pouvoir être composées entre elles sans être retouchées.
Les applications sont des boîtes noires, elles :

- lisent leurs données sur un fichier logique appelé entrée standard (descripteur 0)
- écrivent leurs données sur un fichier logique appelé sortie standard (descripteur 1)

L'association des fichiers logiques aux fichiers physiques peut être modifiée par un processus de redirection.
Une sortie erreur standard est aussi définie (descripteur 2) (par défaut associée au terminal).
Redirection de :

l'entrée standard : commande < nom_fichier_lisible

la sortie standard : commande > nom_fichier_autorisé_en_écriture

la sortie standard (en ajout) : commande >> nom_fichier_autorisé_en_écriture

la sortie erreur standard : commande 2> nom_fichier_erreur

<u>Enchaînement de processus en séquence :</u>
Le déroulement du premier processus n'influe pas sur le déroulement du second (pas d'échanges d'informations) :

```
commande_1 ; commande_2
```

Avec parenthésage pour une redirection :

```
(commande_1 ; commande_2) > nom_fichier
```

Processus concurrents et communiquant entre eux :
Emploi de tubes (encore nommés pipes) : commande_1 |commande_2
Le système assure les téches de synchronisation des écritures et des lectures.
Exemples : `find . -name "*.mp4" | wc -l` : compte tous les fichiers mp4 à partir du répertoire courant
`find / -name "*.txt" | xargs grep ...` recherche d'un motif dans tous les ".txt" du système de fichiers
→ **xargs** commande permettant de récupérer les arguments passés par la commande précédente.

Possibilité de lancer des processus en mode détaché :
l'utilisateur peut continuer à soumettre des commandes à l'interpréte shell qui a lancé la commande : processus en arriére-plan ou en background :

- `commande &`
- `(commande_1; commande_2) &`

La terminaison d'une session tue les processus sauf ceux lancés par l'intermédiaire de la commande **nohup** :
`nohup commande [<données] [>résultats] &`
Les erreurs sont redirigées sur le fichier **nohup.out**.

Lancement à une date donnée :

at	heure:minute am/pm 3_premières_lettres_mois jour `at 9:45 16/09/02 < echo "rentrée des IAO"` les utilisateurs autorisés à utiliser cette command doivent être référencés dans le fichier verb|/etc/at.allow/|
now	un incrément en minutes/hours/days/weeks/months/years est autorisé
batch	dés que le système le permet

Planification de téches :
crontab :

crontab -e appelle l'éditeur de téches à planifier
(export VISUAL=xemacs pour remplacer vi par xemacs sous bash)
format d'une ligne : `minute heure jour_du_mois mois jour_semaine commande`
exemples : `0 0 * * mon find / -atime 7 -exec rm {} \;`
suppression de tous les fichiers non lus depuis une semaine
chaque lundi à 0 heures 0 minutes
crontab -l visualise la planification courante

Remarque : les commandes `nohup`, `at` et `batch` sont autorisées si le démon **cron** existe.

4 Autres commandes à fréquenter :

4.1 Relatives aux interpréteurs de commandes :

Commandes externes : l'exécution de chacune de ces commandes correspond à un processus dédié dont l'exécutable se trouve dans les répertoires **/bin** ou **/usr/bin**.

sh :	appel de l'interpréteur de commandes désuet (mais portable) **sh (Bourne-Shell)**
bash :	appel de l'interpréteur de commandes **bash (Bourne-Shell Again)** (par défaut celui de nombreuses distributions de Linux) l'exécution du processus commence par l'exécution du fichier de nom **.bashrc** → voir la section *Scripts exécutés par l'interpréteur de commandes*
csh :	appel de l'interpréteur de commandes **csh** (shell ayant une syntaxe liée à celle du langage C) l'exécution du processus commence par l'exécution du fichier de nom **.cshrc** (**-f** non-exécution de ce fichier) → voir la section *Scripts exécutés par l'interpréteur de commandes*
export, set ou setenv :	accés aux variables système simples ou d'environnement `export var_env=valeur` pour bash `set var_env=valeur` pour csh/tcsh `setenv var_env valeur` pour csh/tcsh Les variables d'environnement définissent l'environnement de l'utilisateur. Par ex. la variable **PATH** liste les répertoires susceptibles de contenir les exécutables appelés par l'utilisateur (répertoires séparés par :).
printenv ou setenv : **sans arguments**	visualisation des valeurs des variables d'environnement.

4.2 Relatives aux places mémoires occupées :

df -h état des disques logiques (de la partition) notamment nb de blocs libres
du -h espace alloué aux différents fichiers
free état de la mémoire vive
quota affichage de vos quotas et de la place mémoire que vous occupez

4.3 Relatives aux fichiers :

4.3.1 Recherche / classification de fichiers dans le système de fichiers :

find : recherche récursive de fichiers dans une arborescence, par rapport à un nom :
find répertoire -name nom_du_fichier_recherché
exemple : *find . -name "*.txt"* recherche à partir du rép. courant des fichiers d'extension "txt"
si aucun répertoire n'est précisé, la recherche s'effectue à partir de la racine
et d'autres critéres, comme une date de dernière modification
trouver tous les fichiers réguliers modifiés depuis 24 heures :
find répertoire -mtime 1 -type f
trouver tous les fichiers et répertoires ne vous appartenant pas :
find $HOME ! -user $LOGNAME -print
find permet d'exécuter une commande sur tous les fichiers sélectionnés :
find répertoire critéres_de_sélections -exec commande {} \;
{} correspondant au nom du fichier

whereis : recherche du fichier binaire, du fichier source et de la page du manuel d'une commande : *whereis commande*

locate : recherche de tous les fichiers comportant dans leurs noms, une sous-chaîne de caractères
locate sous-chaîne
locate -d répertoire sous-chaîne
Cette recherche exploite un index qui ne peut être créé que par l'administrateur par *locate -u*

file : classification d'un fichier (renvoi de son type par analyse de son contenu)

4.3.2 Recherche de données dans le contenu de fichiers texte :

grep : sélection de lignes satisfaisant un motif particulier (dans un fichier ou entrée std)
-c : nombre de lignes satisfaisant l'expression
-i : pas de distinction minuscules/majuscules
-v : lignes ne satisfaisant pas le motif
-r : recherche récursive à partir d'un répertoire de départ
grep options expression_réguliére fichier

egrep : grep en mieux (utilisation de la norme "Perl" pour les expressions réguliéres)

Compléments sur l'expression réguliére spécifiant le motif de la commande grep :

[début de la définition d'un ensemble de caractères
] fin de la définition d'un ensemble de caractères
- définition d'intervalles
[abc] : soit a, soit b, soit c
[0-9] : un chiffre quelconque
. un caractère quelconque
***** indicateur d'itérations (de 0 à n fois le caractère précédent)
+ indicateur d'itérations (de 1 à n fois le caractère précédent)
. ou .+ : une chaîne de caractères quelconque*
? facultativité (de 0 à 1 fois le caractère précédent)
^ début de ligne en début d'expression ou complément assembliste après un [
[^0-9] : n'importe quel caractère sauf un chiffre
$ fin de ligne en fin d'expression
\ déspécialisation d'un caractère
| alternative entre deux chaînes (`Pierre|Paul`)

Exemples (sur un fichier nommé *départements* contenant des informations (des champs) séparées par des deux-points :

Affichage des lignes ou le second champ est un nombre compris entre 1000 et 4999 :
`grep "^[^:]*:[1-4][0-9][0-9][0-9]:" départements`

Nombre de départements n'appartenant pas à larégion Bourgogne :
`grep -c -v  'Bourgogne$' départements`
(l'emploi des simples quotes évite que le caractère `$` soit interprété par bash)

4.3.3 Visualisations de fichiers avec mise au format :

od : visualisation d'un fichier sous différents formats
-a : les mots sont visualisés en ASCII
-o : les mots sont visualisés en octals
-x : les mots sont visualisés en hexadécimal

xxd -b : visualisation d'un fichier en binaire

tr : substitution ou suppression de caractères
`tr chaîne1 chaîne2`
[...-...] : intervalle, [[:classe:]] (digit, alpha, alphanum, upper, lower)
-d pour la suppression des caractères
tr ab AB fichier

sed : éditeur standard ne travaillant pas en mode interactif fichier ou entrée std
`-e s / expression_réguliére / chaîne_de_remplacement / g` (g=toutes les occurr.)
`sed -e s/a/A/g fichier`

sort : tri les lignes du fichier
-f : minuscules et majuscules identiques
-r : inversement de l'ordre
-u : un seul exemplaire des lignes est conservé
-o : pour préciser le fichier de sortie

uniq : élimination des lignes redondantes successives
-c : chaque ligne est précédée de son nombre d'occurrences

4.3.4 Comparaisons entre fichiers :

cmp : comparaison des contenus de deux fichiers
affichage du numéro de la ligne et du caractère de la première différence
`cmp fichier1 fichier2` → differ : char ..., line ...

diff : affichage de toutes les lignes différentes
diff fichier1 fichier2

comm : affichage sur trois colonnes des différences
lignes appartenant au premier, au second, aux deux
`comm fichier1 fichier2`

4.3.5 Transformations de fichiers :

head : écriture sur la sortie standard des n premières lignes des fichiers (par défaut 10)
`head -n fichier1 ...`

tail : écriture sur la sortie standard des n dernières lignes des fichiers (par défaut 10)
`tail -n fichier1 ...`
lignes 10 à 20 : `head -20 fichier1 | tail -10`

split : découpage en séquence d'un fichier
`split -nombre_de_lignes fichier`
les fichiers produits ont pour suffixes de .aa à .zz

cat : concaténation de plusieurs fichiers en un seul
`cat fichier1 ... fichiern > nouveau_fichier`

cut : sous-ensemble de colonnes d'un fichier
-c : liste de portions (c1-c2 : intervalle, c1- : jusqu'à la fin)
-d : séparateur
-f : dans la cas d'un séparateur, numéros des champs à extraire
`cut -c1-10 départements` (caractères 1 à 10)
`cut -f1,4- -d: départements` (champs 1, 4 et suivants)

4.3.6 Cryptages, compressions et archivages de fichiers :

touch modification de la date de dernière modification d'un fichier

crypt cryptage / décryptage de fichiers
`crypt clé <fichier_en_clair >fichier_encrypté`
`crypt clé <fichier_encrypté`
n'est pas implémenté sur tous les systèmes

gzip compression → suffixe *.z* ou *.gz* associé au fichier compressé
(l'option **k** permet de conserver le fichier originel)
gzip -d décompression
gunzip ou **unzip** idem

La compression de données est surtout associée à lacréation d'**archives** (une archive permet la création d'un seul fichier correspondant généralement au contenu d'une arborescence de répertoires).
Cette compression est par défaut effectuée par la commande **gzip** lors de l'utilisation de la commande d'archivage **tar** avec l'option **z** ou avec la commande **tgz**.

tar anciennement lecture/écriture sur supports magnétiques (disquettes, bandes ...)
archivage (compilation de fichiers en un seul) et désarchivage

lecture : *tar -tvf archive* (d'une archive)
tar -tvf /dev/fd0 (d'un support magnétique)

archivage : *tar -cvf archive fichier_1 ... fichier_n*
tar -cvf archive répertoire_1 ... répertoire_n
→ - après l'option *f* signifie que *tar* lit/écrit sur l'entrée/la sortie standard :
tar -cvf - fichiers >archive

désarchivage : *tar -xvf archive* (fichiers d'une archive)
→ option *z* pour décompresser une archive compressée avec *gzip*
tar -xzvf archive
→ option *k* pour conserver des fichiers préexistants
tar -xkvf archive

tgz méme commande que *tar* mais compresse le résultat (similaire à l'utilisation de l'option **z**)

4.3.7 Impressions de fichiers :

lpr impression
lpr -Pcode imprimante -#nombre_impressions fichier ...
lpq affichage de la queue d'impression
lprm suppression d'un job dans la queue d'affichage

4.4 Montage/démontage de systèmes de fichiers :

Généralement, les périphériques (disques durs externes, clefs... sont automatiquement montés, céd intégrés au système de fichier, quand ils sont connectés par exemple aux slots USB, mais bon quelques fois il faut les monter à lamain, ou par exemple les démonter pour les remonter en lecture seule par précaution.
Ce sont les commandes **mount** et **umount** (pour le démontage) qui permettent de faire cela.

mount: montage d'une arborescence de fichiers
forme générale : *mount device répertoire*
montage d'une clef : *mount /dev/sdb /media/login/nomDeLaClef*
montage en lecture seule d'un disque externe: *mount -r /dev/sdb /monDisque*

Sans paramètres, la commande **mount** liste tous les points de montage, par exemple :
`/dev/sda5 on / type ext4 (rw,errors=remount-ro)`
→ la cinquiéme partition (logique) du premier disque dur (sda5) est montée sur le répertoire /

Le fichier **/etc/fstab** donne également des informations sur les montages effectués.

5 Session de travail distante, transfert de fichiers d'une/sur une machine distante :

5.1 Ouverture d'une session de travail sur une machine distante :

Avec **ssh** : `ssh login@adresseIP`

5.2 Transferts de fichiers entre machines :

- **scp** : copie de fichiers, exemples d'utilisation :
  ```
  scp fichier login@adressseIP:cheminAPartirDurepertoireDAccueil
  scp login@adressseIP:cheminAPartirDurepertoireDAccueil/fichier cheminSurLaMachineLocale
  ```
- **ncftp** et **ftp** sont expliquées plus en détail ci-après.

Pour éviter de s'authentifier systématiquement avec un mot de passe, il est possible de générer une paire de clefs publique et privée. Par exemple pour créer cette paire de clef en utilisant le chiffrement RSA, utilisez la commande suivante : **ssh-keygen -t rsa** (Il vous sera aussi demandé une passphrase (un mot de passe) pour sécuriser la clef privée).
Les clefs seront stockées dans le dossier /**.ssh** sous le nom **id_rsa** (la clef privée) et **id_ras.pub** (la clef publique). La clé publique doit alors être copiée sur le serveur distant dans le dossier /**.ssh/authorized_keys**.

ncftp :	Transfert de fichiers entre la machine locale et distante	
Connexion :	**ncftp machine**	connexion en tant qu'anonyme
	ncftp -u login machine	connexion avec login
Site local :	**l**commande_shell	
	lcd	changement de répertoire
	lls	liste des fichiers
	lmkdir	création d'un répertoire
	...	
Site distant :	**cd**	changement de répertoire
	ls	liste des fichiers
	mkdir	création d'un répertoire
	...	
Transferts	**get *.c**	importation en utilisant le joker "*"
	put *.java	exportation en utilisant le joker "*"
Complétion des noms de fichiers avec <Tab>		

ftp :	Transfert de fichiers sur une machine distante (rendu obsoléte pat *ncftp*)	
Connexion :	*ftp machine*	puis saisie du mot de passe
Site local :	**!**commande_shell	
	lcd	changement de répertoire
Site distant :	**cd**	changement de répertoire
	ls	liste des fichiers
	mkdir	création d'un répertoire

Copie de fichiers de la machine locale vers la machine distante :

put	nom_fichier_local [nom_copie]
append	nom_fichier_local nom_fichier_distant
mput	nom_fichier_local1 ... nom_fichier_local_n

Copie de fichiers de la machine distante vers la machine locale :

get	nom_fichier_distant [nom_copie]
mget	nom_fichier_distant1 ... nom_fichier_distantn
prompt	bascule mode interactif / non interactif

Sessions :

open :	ouverture de la connexion
close :	fermeture de la connexion sans quitter ftp
quit ou **bye** ou **ctrl D** :	fin de ftp

glob (dés)activation du mécanisme d'expansion (* et ?)

5.3 Récupération d'une ressource prise sur internet :

wget ou **cURL** (abréviation de *client URL request library*) sont des commandes qui permettent de récupérer le contenu d'une ressource accessible sur le web.

Exemples avec curl :
curl www.lirmm.fr
curl www.lirmm.fr > pageAccueilLirmm

6 Outil de gestion de versions GIT:

GIT qui a complètement rendu obsolète *Subversion*(**SVN** en abrégé) permet de gérer les différents fichiers des différentes versions d'une application informatique (ou plus globalement de tout corpus de textes). Ces fichiers pourront/devront être répartis sur plusieurs machines (au moins celles des différents contributeurs à l'application et sans doute d'autres serveurs de diffusion).

GIT est à privilégier sur SVN car ce vieil outil centralise l'intégralité des fichiers sur un serveur unique, ce qui rend l'application vulnérable en cas de défaillance de celui-ci.
Nous ne présenterons (briévement) donc que le fonctionnement de GIT (attention cette présentation est vraiment minimaliste).

GIT permet de synchroniser des fichiers qui se trouvent chez vous dans un dépôt "local" vers un dépôt distant et réciproquement. (A proprement parler il n'y a donc pas de serveur et de client...)

Un fichier géré dans un dépôt GIT passe par plusieurs états :

- de votre dépôt vers un dépôt distant : sélectionné (commande **add**) → déposé (commande **commit**) → copié sur un autre dépôt (commande **pull**
- du dépôt distant vers votre dépôt : copié (commande **push**)

Git permet aussi de créer des branches de développement (qui ne sont pas abordées ici) mais a priori, vous n'en aurez pas tout de suite besoin avant de commencer un projet conséquent comprenant plusieurs options de développement !

Voici les commandes de base à utiliser pour :

- créer un **dépôt GitLab** (géré par la Faculté des Sciences) distant qui centralisera les fichiers de votre projet ;
- le cloner sur vos machines locales ;
- et ensuite synchroniser le dépôt Gitlab avec vos dépôts locaux.

(Je considére que vous êtes sous Linux mais cela fonctionne sous n'importe quelle plateforme).

6.1 Création d'une clef de connexion sécurisée sur votre machine locale et installer git :

ssh-keygen (sauver la clef dans `/home/<login>/.ssh/id_rsa`
→ voir la page d'explication : `https://gitlab.etu.umontpellier.fr/help`
→ deux clefs sont créés :
id_rsa.pub : la clef publique qui pourra être publiée
id_rsa : la clef privée
installer git (sous Linux/Ubuntu : **sudo apt-get install git**)

6.2 Création du dépôt Gitlab de la FDS :

Pour utiliser ce service, il suffit de se rendre à l'adresse suivante :
https://gitlab.etu.umontpellier.fr
puis de saisir son adresse institutionnelle UM (prenom.nom@umontpellier.fr) et son mot de passe ENT :
→ cliquer sur le lien "créer une clef ssh" et copier la clef publique dans le formulaire

6.3 Création du dépôt local et synchronisation avec le dépôt distant :

SUR VOTRE MACHINE et dans le répertoire de travail :
cloner votre dépôt : commande à recopier sur le site Gitlab du type :
`git clone git@gitlab.etu.umontpellier.fr:<identifiant de votre projet>`

Puis vous pourrez utiliser les commandes suivantes :
git add <fichier à rajouter dans le dépôt> → par exemple : git add *
git commit -m "commentaire" → mise à jour du dépôt local
git push → mettre à jour le dépôt distant
get pull → mettre à jour le dépôt local

Il est essentiel avant de travailler sur vos codes de réaliser un "git pull" pour récupérer les nouvelles versions de codes.
Nous n'épiloguerons pas sur le cas où lors d'un "git push", GIT vous signale qu'il y a un conflit avec une autre version de code déposé entretemps...

Un fichier **.gitignore** permet de recenser les fichiers à ne pas prendre en compte (quand vous faites un git add *)
git status permet d'avoir à un moment donné l'état de votre dépôt.

Remarque, pour créer ex nihilo un dépôt (sans le cloner), vous devez utiliser la commande **git init** : **git init** → création d'un sous-répertoire **.git**
git config –global user.name "..."
git config –global user.email "...@umontpellier.fr"

7 Editeurs de texte et environnement de programmation :

Tout d'abord, il ne faut pas confondre les éditeurs de texte pour la programmation qui ne rajoutent aucune information aux données enregistrées, aux éditeurs de bureautique comme *LibreOffice* (qu'il ne faut donc pas utiliser pour programmer !).

7.1 Visual Studio Code :

Je vous recommande d'utiliser **Visual Studio Code** : à lancer avec la commande **code .** (ou *codium .*, *vscodium .*)
En effet, il est très puissant et peut être considéré comme un véritable **environnement de programmation**.

7.2 Liste des autres éditeurs "classiques" disponibles :

ed	le plus ancien, cité ici pour des raisons sentimentales
vi	éditeur pleine page (c'est à dire pouvant être directement utilisé dans le terminal)
nano	idem, plus moderne
atom	éditeur graphique
Sublime Text	la star du moment
emacs	ancien, indémodable, complexe mais tellement intéressant (macros ...)

7.2.1 Aperçu de vi :

vi est un éditeur *plein texte*, c'est à dire qui se passe d'interface graphique. Quelquefois, il peut vous sauver la vie.

Cet éditeur possède deux modes :

- le **mode commande** (positionné au lancement) ;
- le **mode insertion**.

Passage du mode commande au mode insertion :

- i : insertion avant la position du curseur
- a : insertion après la position du curseur
- A : insertion à la fin de ligne
- o : insertion après la ligne
- O : insertion avant la ligne

Passage du mode insertion au mode commande : Touche Escape ou F11

Déplacements :
Par les fléches sur certains terminaux

1G	au début du texte
nG	à la nième ligne
G	à la fin du texte
Ctrl P	à la ligne précédente
Return / Ctrl N	à la ligne suivante
Blanc	au caractère suivant
Backspace	au caractère précédent
Ctrl B	à la page précédente
Ctrl F	à la page suivante

Suppressions :

x	suppression du caractère
dd	suppression de la ligne
D	suppression de la fin de ligne

Copier / Coller une ligne :

Y	pour copier
p	pour coller

Sauver les modification : :w
:w nom_fichier

Quitter : :q

7.2.2 Aperçu d'emacs :

Emacs est un éditeur de texte un peu désuet mais que je garde dans mon cœur (pour des raisons obscures).

Liste des raccourcis claviers les plus fréquemment utilisés (C- signifie contrôle) :

<**C-x**> <**C-f**>	pour ouvrir un fichier (existant ou nouveau) le nom du fichier peut être préfixé par un chemin de répertoire quelconque
<**C-x**> <**C-s**>	pour sauver
<**C-x**> <**C-c**>	pour quitter
<**C-s**>	pour chercher une chaîne de caractères en avant dans le texte
<**C-r**>	pour chercher une chaîne de caractères en arriére dans le texte
<**Alt-x**>	pour passer en mode commande ce qui permet par exemple de chercher une ligne avec *goto-line*
<**C-g**>	pour annuler le passage en mode commande

Couper/Copier/Coller :

<**Sélection à lasouris**> <**C-w**>	pour couper
<**C-y**>	pour coller/copier

Subdivision d'Emacs en plusieurs fenêtres :

<**C-x**> **2**	pour créer deux fenêtres horizontales faire <**C-x**> <**C-f**> pour ensuite charger le fichier désiré
<**C-x**> **3**	pour créer deux fenêtres verticales
<**C-x**> **0**	pour détruire la fenêtre où se trouve le curseur actif
<**C-x**> **1**	pour ne garder que la fenêtre où se trouve le curseur actif (bien entendu, une fenêtre est active si vous avez cliqué en son sein)
<**C-x**> **o**	pour passer d'une fenêtre à une autre

8 Introduction au scripting système :

A partir d'un terminal Linux (un *shell*), il est possible de créer et d'exécuter des **programmes système**, c'est à dire des programmes qui vont faire appel à des exécutables (notamment des commandes système) ou exploiter des informations du système d'exploitation. Ces programmes qui sont aussi appelés des **scripts** car ils sont interprétés (et non compilés), peuvent permettre :

- de mémoriser des lignes de commandes complexes et/ou paramétrables (par exemple je veux rechercher à partir d'un répertoire dont le nom est passé en paramètre au script tous les fichiers qui ont une certaine extension (second paramètre du script) et les supprimer) ;
- de créer de nouvelles commandes (par exemple je veux créer une commande nommée *bashConfig* qui m'indique suivant différentes options comment est configuré bash) ;
- de réaliser de manière régulière (par exemple via le planification *crontab*) des sauvegardes ou d'autres opérations ;
- de procéder à des recherches particulières dans le système de fichiers ;
- d'effectuer des analyses de l'état du système, du comportement des utilisateurs ;
- ...

Pour ce faire, plusieurs langages de haut niveau et interprétés sont à votre disposition :

- les langages de programmation associés à Unix/Linux comme **bash** ou *(t)csh* (et d'autres langages de shells...)
- les langages **Python**, *Perl*, *Ruby* (et d'autres)

Les langages **bash** et (t)csh ont le gros avantage de permettre l'interface directe de commandes et les gros inconvenients de ne pas être modulaires, d'avoir des syntaxes luxuriantes et piégeuses (surtout bash). De ces deux langages, seul bash sera abordé en cours car c'est le plus répandu.
Des langages de plus haut niveau et universels (car ils peuvent être utilisés pour d'autres buts que la programmation système), Python, Perl et Ruby, seul Python sera entrevu en cours en perspective de Bash (et sans utiliser le paradigme de la programmation par objets).

Remarques :
Il est bien sûr aussi possible d'utiliser le langage C pour créer des programmes système mais celui-ci, non interprété et de bas niveau, est plus difficile à maîtriser et la création des programmes en sera beaucoup, beaucoup, plus longue. Cela-dit, quand le programme doit interfacer des appels systèmes (les fonctions proposées par le noyau du système), le langage C est souvent choisi (par exemple, vous avez l'idée d'écrire un nouvel interpréteur de commandes...).
Sous Windows, outre Python, Perl et Ruby, le langage phare dans la programmation système est **PowerShell**.

9 Scripting bash :

bash et (t)csh sont des **interpréteurs de commandes** : leur rôle est d'interpréter les lignes de commandes (une ligne de commande pouvant contenir plusieurs commandes séparées par des points-virgules, des pipes...) et saisies par l'utilisateur dans un terminal Unix/Linux.
Ils sont aussi capables d'interpréter des instructions (manipulation de variables, tests, boucles...) et possèdent donc **leurs propres langages de programmation** (mais beaucoup moins confortables que ne peuvent l'étre les langages de programmation **Python**, **Perl** ou **Ruby**).

Un programme qu'il soit écrit pour bash ou (t)csh peut être exécuté de différentes manières :

- **interactivement** : les instructions du programme sont directement saisies dans le terminal
  → un prompt particulier (>) apparaît tant qu'un bloc d'instructions n'est pas fini.

- sous la forme d'**un programme exécutable** (généralement suffixé par **.sh**) : par exemple, le script nommé *bonjour.sh* contenant l'instruction `echo bonjour` sera exécuté par l'une des commandes suivantes :
  - `./bonjour.sh` si le script est exécutable (le point fait référence au répertoire courant) ;
  - `bonjour.sh` si le script est exécutable et si la variable d'environnement **PATH** contient le nom du répertoire où le script se trouve ;
  - `source bonjour.sh` ou `. bonjour.sh` si le script n'est pas forcément exécutable ;

  – `bash bonjour.sh` pour exécuter ce script par un autre processus bash que celui qui gère le terminal (car vous avez des raisons particulières pour faire cela...).

Un programme interprétable par l'interpréteur de commandes bash ou (t)csh est appelé un **script shell**, Par défaut il est exécuté par l'interpréteur de commandes actif (celui dont le nom apparaît dans la variable d'environnement SHELL).

Si vous voulez explicitement préciser dans un script shell, quel interpréteur de commandes doit l'exécuter, vous pouvez faire apparaître un shebang `#!` sur la première ligne du script :

- `#!/bin/csh` pour **csh** (ou tcsh)
- `#!/bin/bash` pour **bash**

9.1 La configuration de bash (et de (t)csh) :

bash ou (t)csh sont configurables à différents niveaux, en pratique seule la configuration au niveau de l'utilisateur nous intéressera.

Voici les principaux fichiers de configuration exécutés sous **bash** :

- */etc/profile*
- *.bash_profile* (dans votre répertoire d'accueil)
- **.bashrc** (dans votre répertoire d'accueil) → ce fichier va contenir vos configurations

Ces fichiers de configuration vous permettent de :

- compléter la variable d'environnement **PATH** qui liste tous les répertoires dans lesquels des exécutables peuvent être retrouvés par l'interpréteur de commandes. Par exemple, voici comment la compléter avec le répertoire courant : *export PATH=$PATH:.*

- de définir des alias qui sont des synomymes à une commande :
  alias nouvelle_commande='ancienne_commande'
  alias c='clear'
  Pour créer des alias avec paramètres, il faut créer des fonctions :
  exemple : `alias chm='function _chm() { chmod $1 $2; }; _chm'`

- changer la valeur du prompt :
  – Sous bash, il faut insérer des codes spéciaux dans la valeur de la variable **PS1** :
    * \d : date
    * \h : nom de la machine
    * \W : nom du répertoire courant
    * \w : chemin du répertoire courant
    * \u : login de l'utilisateur

    Exemple avec *PS1='\W \d $ '*

9.2 Variables et paramètres :

9.2.1 Les variables système :

Des variables prédéfinies permmettent d'accéder (et de modifier) des informations relatives au système d'exploitation :

- les **variables système d'environnement** dont les valeurs seront accessibles par les programmes lancés à partir de l'interpréteur de commandes ;
- les variables système (simples) dont les valeurs n'ont une importance que pour l'interpréteur de commandes lui-méme.

Voici les principales variables d'environnement système :

DISPLAY :	adresse_IP:numéro_du_serveur_X.numéro_d_écran où les applications X vont s'afficher
HOME :	répertoire d'accueil
LOGIN :	login
PATH :	liste des répertoires où le shell cherche les exécutables à exécuter
PWD :	répertoire courant
SHELL :	shell (par exemple /bin/bash)
TERM :	type de terminal (par exemple xterm)

Pour modifier une variable d'environnement, il faut utiliser la commande **export**.
Exemple (ici pour rajouter le répertoire courant dans le PATH) :

```
export PATH=$PATH:.
```

La principale variable système "simple" est celle qui donne au valeur au prompt dans le terminal : **PS1** sous *bash*) .

9.2.2 Déclaration / initialisation des variables :

Les variables scalaires (simples) n'ont pas besoin d'être déclarées et sont considérées par défaut comme étant des chaînes de caractères à moins d'être "injectées" dans des expressions arithmétiques.
Les variables composites (tableaux simples et tableaux associatifs) peuvent être déclarés avec l'instruction **declare**

```
nom_variable=valeur
```

Exemple :

```
prenom=Pierre
```

(Attention : n'insérez pas d'espace avant ou après le signe =)

9.2.3 Affichage d'une variable :

Une fois définie, une variable doit être préfixée par **$** pour être utilisée !
La valeur d'une variable est affichée dans le terminal grâce à l'instruction **echo**, par exemple :

```
echo $prenom
```

Il est aussi à noter qu'une chaîne de caractères peut être délimitée ou non par des simples ou des doubles quotes, par exemple les trois instructions suivantes sont équivalentes :

```
echo chaine
echo 'chaine'
echo "chaine"
```

L'instruction *printf* qui existe aussi, ne sera pas évoquée.

9.2.4 Récupération d'une valeur au clavier :

Il faut utiliser la commande **read** :

```
read entree
echo $entree
```

9.2.5 Les tableaux simples (array) :

Un tableau simple (*array*) qui regroupe des valeurs indicés par un indice débutant à 0 peut être défini ainsi :

```
nomDutableau=(valeur1 valeur2 ...)
```

Exemple :

```
planetes=(Mercure Vénus Terre Mars Jupiter Saturne Uranus Neptune)
```

Si le tableau est construit progressivement, il est préférable de le déclarer (mais ce n'est pas obligatoire) :

```
declare -a planetes
planetes[0]=Mercure
planetes[1]=Vénus
...
```

Pour afficher un élément d'un tableau, il faut non seulement utiliser $ et indiquer son indice, mais aussi encadrer le nom de l'élément par des accolades pour que Bash puisse bien identifier celui-ci :

```
echo ${planetes[0]}
```

Pour afficher (d'un seul coup) tous les éléments d'un tableau, il faut utiliser l'indice virtuel :

```
echo ${planetes[@]}
```

Pour afficher un slice (c'est à dire une partie continue des éléments d'un tableau, il faut utiliser l'indice virtuel et une ou deux bornes. Par exemple, l'instruction suivantes affichera les éléments du tableau à partir du second :

```
echo ${planetes[@]:1}
```

9.2.6 Les tableaux associatifs :

Les tableaux associatifs permettent d'associer des clefs à desvaleurs (c'est à dire que l'indice numérique du tableau simple est remplacé par une chaîne de caractères) :
Il faut les déclarer par `declare -A`

```
declare -A jours
jours=([dimanche]=sunday [lundi]=monday)
```

Pour afficher (d'un seul coup) tous les éléments d'un tableau, il faut utiliser l'indice virtuel et faire précéder le nom du tableau par un point d'interrogation si on veut accéder aux clefs :

Affichage des valeurs :

```
echo ${jours[@]}
```

Affichage des clefs :

```
echo ${!jours[@]}
```

9.2.7 Valuation d'une variable par le résultat d'une commande :

Une variable peut être valuée par le résultat d'une commande suivant deux syntaxes (préférez la seconde plus moderne) :

```
variable=`commande`
variable=$(commande)
```

Exemple :

```
ll=$(ls -l)
```

9.2.8 Calcul arithmétique :

Un calcul arithmétique peut être effectué suivant deux syntaxes possibles :

- avec la syntaxe `$((<opération>))` : `i=$((3+4)); echo $i`
- avec la commande interne `let` : `let i=3+4; echo $i`

9.2.9 Les paramètres de la ligne de commande :

`$*` et `$@` permettent d'accéder aux paramètres de la ligne de commande.
`"$@"` permet de gérer des paramètres qui comprendraient des espaces (cas assez rare).

nombre de paramètres	`$#`
liste des paramètres	`$*` ou `$@` ou `"$@"`
nom du script	`$0`
niéme paramètre (raccourci $n)	`$n`

9.3 Structures conditionnelles :

9.3.1 Structure conditionnelle avec if :

Sous *bash* et comme dans la quasi totalité des langages de programmation, le mot réservé **if** introduit une expression conditionnelle dans une structure de programmation qui permet de conditionner l'exécution d'un bloc d'instructions. Mais à la différence des autres langages de programmation, l'expression conditionnelle peut être encadrée par différents

délimitateurs (crochets, doubles crochets, doubles parenthèses) ou méme ne pas en utiliser : cette diversité est déconcertante et rend la programmation en bash assez piégeuse. Les crochets sont des alias pour la commande **test**.

Voici en apéritif, quelques exemples d'expressions conditionnelles :

```
if ls | grep -q Cours ->     l'expression conditionnelle est une ligne de commande qui sera validée
                             si dans le répertoire courant le mot "Cours" apparaît
                             (l'option q de grep l'empéche d'afficher son résultat)
if test $i -gt 0          -> comparaison arithmétique avec la commande test et les opérateurs classiques
                             ici -gt teste qu e le premier opérande est plus grand que le second
if [ $i -gt 0 ]           -> les crochets remplacent la commande test
                             il faut mettre une espace après le premier crochet et avant le dernier
if [[ $i > 0 ]]           -> Les doubles crochets permettent d'utiliser tous les opérateurs arithmétiques
if [[ $a > $b ]]          -> PROBLEME : si $a vaut 20 et $b vaut 100, ce test renvoie vrai car 2 est supé
if (( $a > $b ))          -> avec les doubles parenthèses, la comparaison arithmétique fonctionne car les
```

Revenons au schéma d'utilisation de la structure conditionelle avec if (et ici en encadrant l'expression conditionnelle avec les doubles parenthèses ce que nous ferons quand nous voudrons tester la valeur d'une variable numérique) :

```
if (( expression conditionnelle ))
then
        bloc d'instructions
else                                (bloc facultatif)
        bloc d'instructions
fi
```

Attention si vous ne passez pas à la ligne avant *then*, *else* ou *fi*, vous devez rajouter un point-virgule pour le signaler à l'interpréteur bash (c'est affreux...) !

```
if (( expression conditionnelle )); then bloc d'instructions; fi
```

9.3.2 Utilisation des simples ou doubles crochets :

La commande **test** ou les **simples crochets** délimitant l'expression conditionnelle permettent :

- de tester le statut des fichiers avec des opérateurs unaires (*)
- de mettre en place des comparaisons arithmétiques sur des opérandes numériques (**)
- de tester l'existence d'un élément de tableau

(*) Voici les opérateurs les plus populaires de test sur les statuts de fichiers :

d type répertoire
f type ordinaire (fichier régulier)
e existence du fichier
o propriétaire du fichier
r droit de lecture
w droit d'écriture
x droit d'exécution
z taille nulle

Exemple :

```
if [ -f $fichier ]
then
   echo $f est un fichier régulier
fi
```

(**) Voici les opérateurs "vintage" de tests arithmétiques :

-eq égalité
-ne différence
-lt plus petit que
-gt plus grand que
-lt plus petit ou égal
-ge plus grand ou égal

Les **doubles crochets autour** de l'expression conditionnelle permettent :

- d'avoir le même comportement que les simples crochets
- d'utiliser des opérateurs binaires modernes : ==, !=, <, <=, >, >=, ainsi que l'opérateur de négation !
  mais attention, si la comparaison se fait sur des chaînes de caractères dont les valeurs sont des suites de chiffres, la comparaison reste lexicographique.

Les expressions conditionnelles peuvent intégrer le *ou* (double barre verticale) ou le *et* (double esperluète) logique :
Exemple (test si un fichier dont le nom est contenu dans la variable `$f` et se termine par la chaîne passée en paralétre est régulier) :

```
if [[ "$f" == *$1 ]] && [[ -f "$f" ]]
do
   ...
done
```

9.3.3 Synthèse des différents emplois de la structure conditionnelle if :

En résumé, il est donc possible :

- d'utiliser la commande *test*, des simples ou des doubles crochets en encadrement de l'expression conditionnelle, pour effectuer :
  - des tests via des opérateurs unaires, exemple : `if [ -f $fichier ]` → teste si le fichier est un fichier régulier
  - des tests via des opérateurs binaires ("vintage" ou modernes)
- de ne pas encadrer l'expression conditionnelle dans le cas de l'évaluation directe du résultat d'une commande :
  exemple : `if grep -q $1 $fichier` (recherche du premier paramètre donné au script dans le contenu d'un fichier)

Des exemples d'emplois de diverses structures conditionnelles sont donnés dans le paragraphe *Exemples de scripts bash*).

9.3.4 Structure conditionnelle avec case :

```
case expression in
    pattern1 )
         bloc d'instructions ;;
    pattern2 )
         bloc d'instructions ;;
    ...
esac
```

9.4 Structures itératives :

9.4.1 Structure itérative avec while :

La structure itérative *while* (tant que) permet de répéter l'exécution d'un bloc d'instructions tant qu'une condition est vraie :

```
while (( expression conditionnelle avec opérandes numériques ))
do
    bloc d'instructions
done
```

Exemple :

```
i=1
while (( $i < 10 ))
do
    echo $i
    i=$((i+1))
done
```

9.4.2 Structure itérative avec for

```
for variable in séquence
do
    bloc d'instructions
done
```

La séquence peut être :

- une suite de valeurs espacées → `for i in 1 2 3; do echo $i; done` → `for i in "mot1 mot2 mot3"; do echo $i; done`
- les éléments d'un tableau simple ou associatif → voir les exemples donnés ci-après
- le résultat **produit par l'exécution d'une commande** → `for i in $(cat fichier); do echo $i; done`

Affichage, un par un, des éléments d'un tableau simple :

```
tab=(e1 e2 e3)
for i in ${tab[@]}
do
   echo $i
done
```

Affichage, un par un, de la clef puis de la valeur de chaque élément d'un tableau associatif :

```
declare -A jours
jours=([dimanche]=sunday [lundi]=monday)

for clef in "${!jours[@]}"
do
   echo $clef : ${jours[$clef]}
done
```

9.5 Lecture de fichiers :

La lecture d'un ficher est généralement effectué via la commande *cat*. Voici une structure de programmation classique (mais qui est imparfaite) :

```
for ligne in $(cat $fichier)
do
    echo $ligne
done
```

Le probléme est que chaque mot du fichier (et non chaque ligne) est recopié dans la variable *ligne*. Il existe en effet une variable nommée **IFS** (l'"Internal Field Separator") qui spécifie les caractères de "découpe" et comprend par défaut l'espace, la tabulation et le newligne (`\n`). Il faut donc au préalable la redéfinir :

```
IFS=$'\n'
for ligne in $(cat $fichier)
do
    echo $ligne
done
```

9.6 Fonctions :

Il est possible de créer des fonctions :

```
function nomDeLaFonction(paramètres)
{
    bloc d'instructions
}
```

9.7 Pour forcer l'arrét d'un script et récupérer son "statut" :

- **exit** entier : sortie du script avec instanciation d'une variable système
- $? : valeur retournée par le dernier processus

9.8 Quelques exemples de scripts bash :

9.8.1 Affichage de tous les répertoires du répertoire courant :

```
for i in *
do
     if [[ -d $i ]]
     then
          echo $i est un répertoire
     fi
done
```

9.8.2 Affichage des fichiers réguliers qui contiennent la chaîne passée en paramètre du script :

première solution en utilisant une variable intermédiaire et via un comptage :

```
for f in *
do
    if [[ -f "$f" ]]
    then
       r=`grep -c $1 "$f" 2> /dev/null`
       if (( $r > 0 ))
       then
          echo $f contient $1 $r fois
       fi
    fi
done
```

Vous remarquerez l'utilisation des doubles quotes autour de $f pour se prémunir des noms de fichiers avec espaces.

Seconde solution plus frustre en évaluant directement grep dans l'expression conditionnelle :

```
for f in *
do
    if [[ -f "$f" ]]
    then
       if grep -q $1 "$f" 2> /dev/null
       then
          echo $f contient $1
       fi
    fi
done
```

Vous remarquerez l'utilisation de l'option q (*quiet*) de grep qui permet de lui faire renvoyer soit faux ou vrai.

9.8.3 Affichage numérotà deslignes d'un fichier dont le nom est passé en paramètre :

```
IFS=$'\n'
for ligne in $(cat $1)
do
    i=$((i+1))
    echo $i $ligne
done
```

Vous remarquerez avec jubilation :

- la modification du "Internal Field Separator" pour éviter que echo ne réalise des passages à la ligne sur les espaces
- l'incrémentation de la variable i via l'évaluation arithmétique opérée par les doubles parenthèses

Webographie :

- **Pratique de Unix/Linux (commandes) :**
  - **http://juliend.github.io/linux-cheatsheet/** → c'est trés raffraîchissant
  - **http://doc.ubuntu-fr.org/tutoriel/console_commandes_de_base**

- **Installation de Linux/Ubuntu :**
  - **http://doc.ubuntu-fr.org/**
  - **http://linux.developpez.com/tutoriels/debuter-installation/guide-linux-distribution/**

- **Virtuosité en expressions réguliéres :**
  Mastering Regular Expressions 3e (en anglais) - Jeffrey E F Friedl

  Par ailleurs, voici un site de mémos fantastique sur à peu prés tout (enfin en informatique) :
  http://www.cheat-sheets.org/

Index

.bashrc, 9, 13
/etc/apt/sources.list, 7
/etc/fstab, 16
$DISPLAY, 23
$HOME, 23
$LOGIN, 23
$PATH, 13, 22, 23
$PWD, 23
$SHELL, 22, 23
$TERM, 23

alias, 22
Alt F2, 8
Android, 4
apropos, 9
apt, 7
apt-get, 7
archive, 16
at, 13
atom, 19

bash, 13, 21
bash ., 21
bash .bashrc, 22
bash : simples ou doubles crochets, 25
bash $, 24
bash $((...)), 24
bash $(...), 27
bash $*, 24
bash $0, 24
bash $?, 28
bash $n, 24
bash bash_profile, 22
bash case, 26
bash declare, 23
bash echo, 23
bash exit, 28
bash export, 23
bash fonction, 27
bash for, 27
bash if, 24
bash IFS, 27, 28
bash let, 24
bash paramètres, 24
bash profile, 22
bash PS1, 22, 23
bash read, 23
bash source, 21
bash tableau associatif, 24
bash tableau simple, 23
bash Tabulation, 9
bash test, 26
bash while, 26
batch, 13
bureau, 8

cat, 12, 15
cd, 12
chargeur, 8
chgrp, 12
chmod, 12
chown, 12
close, 17
cmp, 15
code, 19
comm, 15
console graphique, 8
console textuelle, 8
cp, 12
cron, 13
crontab, 13
crypt, 16
csh, 13
cURL, 18
cut, 15

date, 9
deb, 4
Debian, 4, 5
device, 11
df, 14
DHCP, 6
diff, 15
dpkg, 7
du, 14

egrep, 14
emacs, 8, 19, 20
emacs coller/copier/couper, 20
export, 13
Expression réguliére, 14

fdisk, 6
fichier passwd, 8
file, 14
find, 13, 14
finger, 10
Firefox, 8
free, 14
ftp, 17
ftp get, 17
ftp lcd, 17
ftp lls, 17
ftp lmkdir, 17
ftp mget, 17
ftp mput, 17
ftp prompt, 17
ftp put, 17

gid, 8, 10
GIT, 18
GIT .gitignore, 19
GIT dépôt GitLab, 18
GIT git add, 18

GIT git commit, 18
GIT git init, 19
GIT git pull, 18, 19
GIT git push, 18, 19
GIT git status, 19
gksudo, 6
GNOME, 4, 8
grep, 14
gunzip, 16
gzip, 16

head, 15
hostname, 9

id, 10
interpréteur de commande, 21
interpréteur de commandes, 8
ip a, 10

KDE, 4, 8
kill, 11
killall, 11

last, 10
lcd, 17
LDAP, 8
LibreOffice, 19
ln, 12
locate, 14
login, 8
lpq, 16
lpr, 16
lprm, 16
ls, 12
lsb_release -a, 9
lscpu, 9
lspci, 9

make, 7
make check, 7
make install, 7
man, 9
mkdir, 12
mode détaché, 13
more, 12
mount, 16
mv, 12

nano, 19
ncftp, 17
netstat -antup, 10
NIS, 8
nohup, 13
now, 13

od, 15
open, 17

parted, 6
PID, 10
pipe, 13
port, 10
PowerShell, 21
PPID, 10
printenv, 13
processus, 10
processus en arriére-plan, 13
ps, 11
pwd, 12

quota, 14

rm, 12
rmdir, 12
RPM, 4
rusers, 10
rwho, 10
Répertoire /bin, 11
Répertoire /dev, 11
Répertoire /etc, 11
Répertoire /home, 11
Répertoire /tmp, 11
Répertoire /usr/bin, 11
Répertoire /usr/local/bin, 11

scp, 17
script shell, 22
sed, 15
service, 10
service –status-all, 10
set, 13
setenv, 13
sftp, 17
sh, 13
sort, 15
split, 15
ssh, 17
ssh-keygen, 18
startx, 8
STAT, 11
SublimeText, 19
Subversion, 18
sudo, 6
sudo apt install, 7
sudo apt update, 7
sudo apt upgrade, 7
sudo apt-get install, 7
sudo do-release-upgrade, 7
SVN, 18
Synaptic, 5, 7

tail, 15
tar, 16
terminal, 8
test, 24
tgz, 16
TIME, 11
top, 11
touch, 16
tr, 15
TTY, 10

tube, 13

Ubuntu, 4, 5
uid, 8, 10
umask, 12
uname -a, 9
uniq, 15
Unity, 4, 8

vi, 19
VirtualBox, 6
Visual Studio Code, 19

Wayland, 4
wet, 18
whereis, 14
who, 10
whoami, 10
Windows 10, 4
WSL, 5
WSL 2, 5

X Window, 4
x2go, 5
xargs, 13
xemacs, 8
xxd, 15