Commandes UNIX sous LINUX

David Gorisse
janv 2009

Plan

- Introduction
- Shell
- Gestion des processus
- Système de fichier
- liste de commandes
- ScriptShell
- MakeFile
- Editeur Texte

Introduction

- Définition Système d'exploitation
- Historique UNIX
- Licence Linux
- Structure de Linux

Système d'exploitation

- **Définition:** Logiciel qui fournit un environnement d'exécution pour les programmes qui vont s'exécuter sur un ordinateur.
- Il doit **gérer les ressources** que vont se partager les programmes :
  - Gestion du processeur
  - Gestion de la mémoire centrale
  - Gestion du système de fichiers
  - Gestion des périphériques
- Un ordinateur ne peut pas fonctionner sans système d'exploitation
- Un ordinateur peut fonctionner avec plusieurs systèmes : Unix, Windows, ...

Historique UNIX

1969 Ken Thompson écrit Unix en assembleur dans les laboratoire de Bell (AT&T)

1973 Dennis Ritchie et Ken Thompson réécrive Unix en langage C (seulement 10% en assembleur)

1974 Unix est distribué librement aux universités américaines

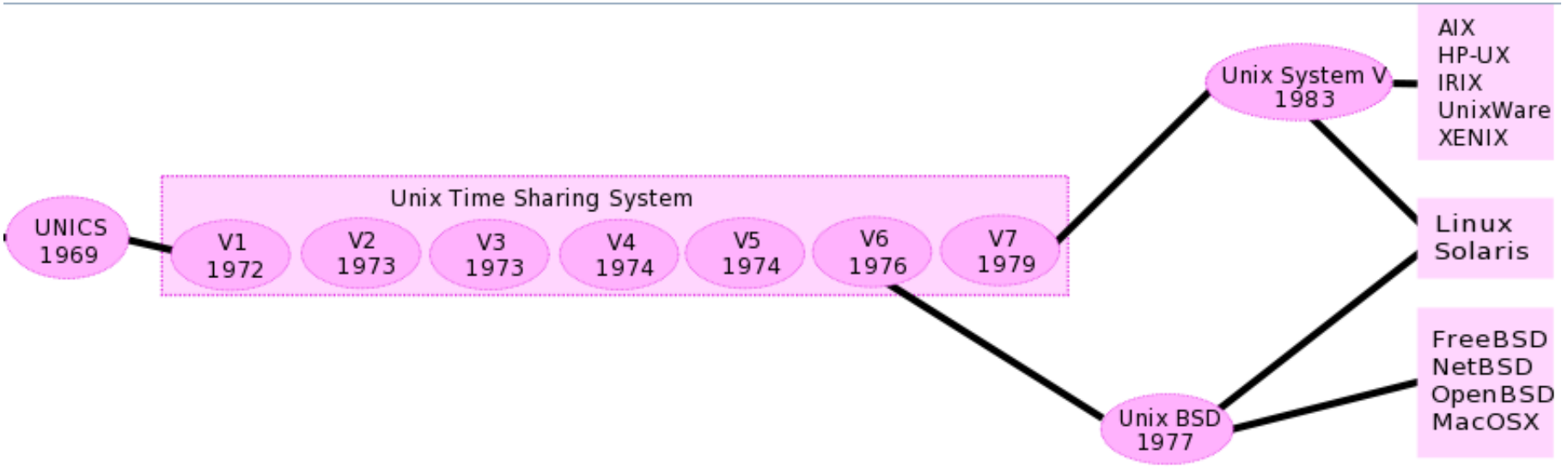
1977 Unix BSD (Berkeley Software Development) 1.0 de l'université de Berkeley aux Etats-Unis

1978 Brian Kernighan et Rob Pike (chercheurs des laboratoires Bell) présentent la philosophie de conception d'Unix

1979 Unix Version 7.0 Commercialisation d'une licence binaire (moins onéreuse que la licence pour le code source)

1983 Unix System V qui a servi de base à la norme POSIX

UNIX-LIKE



1991 Linus Torvalds écrit **LINUX** en s'inspirant de **Unix Sytem V** et de **Unix BSD**

La licence GPL de GNU

- Code Source de **LINUX** est accessible **gratuitement**.
- Qui protège LINUX ?
  - la fondation FSF (Free Software Foundation) a développé un projet GNU pour permettre la distribution de programmes exempts de droits.
- Comment protéger LINUX ?

  => les utilisateur GNU sont soumis à une licence d'utilisation : GPL (General Public Licence) qui spécifie les dispositions légales vis-à-vis de l'utilisation, la distribution et les modifications.
  - ex : toute modification doit être rendu public

Composition d'un système LINUX

3 composants :

- le noyau (Kernel) : partie centrale d'Unix
- un serveur X : interface graphique permettant de créer des applications graphiques.
- un environnement graphique, comme : KDE, Gnome, ...

Le Noyau

- permet de faire le lien entre les programmes et le matériel.

  en mettant à disposition des procédures appelées *primitives*
  ex: demande de plus de mémoire vive

- se charge des tâches essentielles à la bonne marche du système:
  - gestion du système de fichier
  - du processeur
  - de la mémoire centrale
  - des échanges entre l'unité centrale et les périphériques (entrées-sorties)
- partage les ressources rendant le système multi-tâche

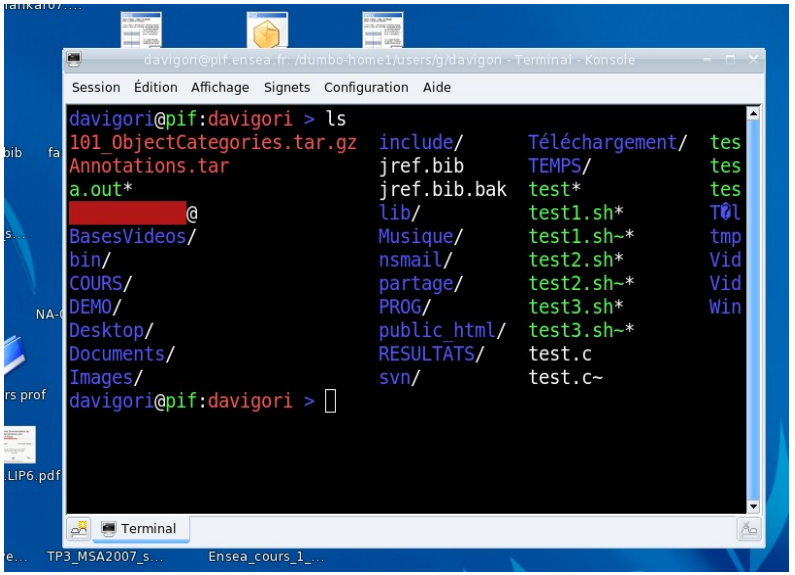
Shell

Shell

- Définition
- Généralité sur les commandes
- Raccourcis
- Caractères spéciaux

Shell

- **point du vue visuel:** écran noir avec en début de ligne un prompt. On peut taper à la suite des lignes de commandes.



- **point de vue informatique:** C'est la liaison la plus élémentaire entre l'utilisateur et le système d'exploitation. Il permet d'interpréter les commandes que l'on entre et d'exécuter des applications.
- **le prompt:** contient le nom de l'utilisateur courant suivit du nom de la machine et du nom du répertoire courant.

  ex: davigori@pif:davigori
- Synonymes de Shell : **console**, **terminal**

syntaxe générale des commandes Unix

```
davigori@pif:UNIX > commande -options arguments
```

- Les *arguments* indiques sur quels objets la commande va agir et les *options* des variantes dans l'exécution de la commande.
  - Les options sont précédées d'un tiret « - ».
  - L'ordre des options est indifférent.
  - Si pas d'arguments, argument par défaut.
- *exemple :* `> ls` `> ls -l` `> ls -al` `ls unix.pdf` `> ls -i unix.pdf`
- *Pour connaître l'utilisation d'une commande : commde --help*

  Ou utiliser le manuel : man commande

syntaxe générale des commandes Unix

- Pour connaître l'utilisation d'une commande :
  - *commde --help*

```
davigori@pif:UNIX > ls --help
Usage: ls [OPTION]... [FICHIER]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort.

Les arguments obligatoires pour les options de formes longues le sont aussi
pour les options de formes courtes.
  -a, --all                  ne pas ignorer les entrées débutant par .
  -A, --almost-all           ne pas inclure dans la liste . et ..
      --author               avec -l, afficher l'auteur de chaque fichier
```

  - Ou utiliser le manuel *: man commande*

ouvre une page du manuel
- utiliser les flèches pour parcourir la page
- « space » pour passer à la fenêtre suivante
- « g » pour revenir au début
- « G » pour atteindre la fin
- « q » pour quitter et revenir au terminal

```
LS(1)                    Manuel de l'utilisateur Linux                    LS(1)

NOM
       ls, dir, vdir - Afficher le contenu d'un répertoire.

SYNOPSIS
       ls [options] [fichier...]
       dir [fichier...]
       vdir [fichier...]

DESCRIPTION
       La commande ls affiche tout d'abord l'ensemble de ses arguments
       fichiers autres que des répertoires. Puis ls affiche l'ensemble des
       fichiers contenus dans chaque répertoire indiqué. Si aucun argument

OPTIONS POSIX
       -C     Présenter les fichiers en colonnes, triés verticalement.

       -F     Ajouter un caractère à chaque nom de fichier pour indiquer son
```

Raccourcis

- la flèche du haut et celle du bas :

  permet de naviguer dans l'historique des commandes

- la touche **TAB**:

  permet de compléter le nom du fichier à condition que le nom soit unique. Sinon, un deuxième appui affichera la liste des possibilités.

- la touche **SHIFT** + flèche haut-bas :

  permet de faire défiler le contenu

- la touche **CTRL**:

  + C : permet d'interrompre une commande en cours d'exécution.

  + D : ferme le terminal en cours.

  + Z : met en attente le dernier processus (met le processus en sommeil)

Caractères spéciaux

- « **?** » et « ***** » sont des caractères des substitutions.
  - « ? » permet de remplacer une lettre unique.
  - « * » permet de remplacer plusieurs lettres.
  - ex : « uni* » représente tous les mots commençant par uni

```
davigori@pif:UNIX > ls uni*
unix.odp  unix.pdf  unix.ps
```

- « [chaine] » représente n'importe quel caractère compris entre [ ]

  [0-9] désigne un chiffre quelconque.

  [^0-9] désigne n'importe quel caractère qui n'est pas un chiffre.

  [!e] désigne n'importe quel caractère qui n'est pas un « e ».

  [-f] désigne « - » ou « f »

  [ac0-1A-Z] désigne « a » ou « c » ou un chiffre ou une lettre majuscule

- ex : rm *[ab] effacera tout fichier du répertoire courant finissant par « a » ou « b »

Gestion des processus

Gestion des processus

- Définition
- État des processus
- Lister les processus
- Interagir avec les processus
- Gestion des jobs
- Reprendre la main sur la console

Définition

- multi-tâche: linux peut gérer plusieurs processus en parallèle
- processus: instance de programme s'exécutant à un instant donné ainsi que son contexte.
- contexte: l'ensemble des ressources utilisées par le programme pour pouvoir se dérouler comme par exemple la mémoire ou les fichiers ouverts.
- PID: numéro unique donné par le système permettant d'identifier un processus. Le propriétaire du processus peut interagir avec ce dernier grâce à cet identifiant.

État des Processus

- principaux états d'un processus:
  - Actif: processus réalisant des actions. Il possède le processeur et réalise des opérations nécessaires à son déroulement.
  - Exécutable: processus en cours d'exécution mais attendant que le processeur soit libre pour réaliser des opérations. Le multi-tâche n'est qu'une illusion, une machine contenant qu'un processeur ne peut réaliser qu'une opération à la fois.
  - Endormi: processus inactif attendant une condition pour redevenir exécutable
  - Zombie: processus terminé qui n'a plus de raison d'exister. Seulement, le système conserve des informations concernant se processus pour divers raisons. Par exemple pour que le processus parent qui l'a créé puisse voir que ce processus est fini.

Lister les processus

- PS : donne une vision instantanée des processus courant du système

l'option -o permet de spécifier les informations à afficher

```
davigori@pif:UNIX > ps -o "%p %P %u %c"
  PID  PPID RUSER    COMMAND
 3671  3642 davigori bash
 6899  3671 davigori soffice
 6914  6899 davigori soffice.bin
26433  3671 davigori ps
```

l'option *-e* permet d'afficher les processus du propriétaire sur la machine

```
davigori@pif:UNIX > ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 init
    2 ?        00:00:00 migration/0
    3 ?        00:00:00 ksoftirqd/0
```

l'option -a permet d'afficher les processus de tous les utilisateurs

Lister les processus

- TOP : donne un aperçu temps réel des processus tournant sur le système
  - montre les processus en ordre décroissant d'utilisation du processeur
  - indique l'utilisation mémoire

=> permet de connaître les programmes consommateurs de ressources

```
davigori@pif:UNIX > top
top - 17:54:16 up 2 days,  9:01,  1 user,  load average: 0.36, 0.39, 0.34
Tasks: 127 total,   1 running, 125 sleeping,   0 stopped,   1 zombie
Cpu(s):  1.0%us,  0.2%sy,  0.0%ni, 98.7%id,  0.0%wa,  0.2%hi,  0.0%si,  0.0%st
Mem:   1017984k total,   926860k used,    91124k free,        0k buffers
Swap:  2096472k total,    65928k used,  2030544k free,   275448k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3151 root      15   0  453m  54m 3392 S    2  5.5  14:31.30 X
 3642 davigori  15   0  143m  17m  10m S    0  1.7   0:06.53 konsole
26938 davigori  15   0 15840 1396 1020 R    0  0.1   0:00.02 top
    1 root      16   0  3700  520  484 S    0  0.1   0:00.76 init
    2 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/0
```

Interagir avec les processus

- KILL : envoie un signal à un processus permettant de modifier le comportement du processus

syntaxe : kill -signal PID

- un signal peut être désigné par son nom ou sa valeur numérique

nom	valeur	description
SIGINT	2	met le processus en pause (CTRL+z)
SIGKILL	9	arrête le programme
SIGSEGV	11	signal envoyé à un programme si le programme tente d'accéder à un endroit mémoire invalide

Gestion des jobs

- **jobs**: tâche demandé par l'utilisateur. Une tâche peut être composée de plusieurs processus. Ce qui les regroupe est appelé un **tube**. ex : `davigori@pif:~ > ls | less`

  ici 2 processus sont créés mais qui sont vu comme un tout : un job

```
davigori@pif:UNIX > jobs
[1]   Running                 ooffice Introduction.odp &
[2]+  Stopped                 ls -F --show-control-chars --color=auto | less
[3]-  Running                 acroread &
```

  [ ] : numéro du job ; + : job le plus récent ; - : avant dernier job ;
  Stopped : job en sommeil (ne peut plus réaliser aucune action)

Reprendre la main sur la console

- Quand un job est lancé, on ne récupère le terminal qu'une fois le job terminé ou mis en sommeil.

```
davigori@pif:UNIX > xemacs
```

- On peut lancer un job en arrière plan en utilisant « & »

```
davigori@pif:UNIX > xemacs&
[3] 29366
davigori@pif:UNIX >
```

cela permet de garder la main sur la console et ainsi lancer d'autres jobs

- Si on oublie le « & », on peut récupérer la main en mettant le job en sommeil : CTRL + z

```
davigori@pif:UNIX > xemacs
[3]   Done                    xemacs

[4]+  Stopped                 xemacs
davigori@pif:UNIX >
```

- Le job n'est plus accessible, il est en sommeil !!! on peut le réveiller en le mettant en arrière plan avec la commande **bg**. (inversement **fg** permet de rappeler le job au premier plan)

Système de fichiers

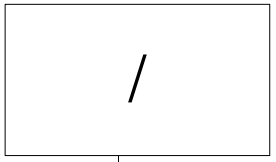
Système de fichiers

- Définition
- Arborescence
- Types de fichiers
- Gestion des droits
- Espace Disque

Définition

- fichiers Unix sont enregistrés dans un ensemble structuré hiérarchiquement en arbre, appelé système de fichiers.
- Un système de fichier :
  - une racine : « / ».
  - des noeuds = fichiers répertoires.
  - des fichiers ordinaires contenant des données ou des programmes.
- Plusieurs systèmes de fichiers sont « montés » sur le système « racine », celui qui contient le répertoire « / ». Ces systèmes sont enregistrés sur des disques physiques différents ou sur des partitions différentes. Mais, tous ces systèmes sont vus par l'utilisateur comme un seul système.

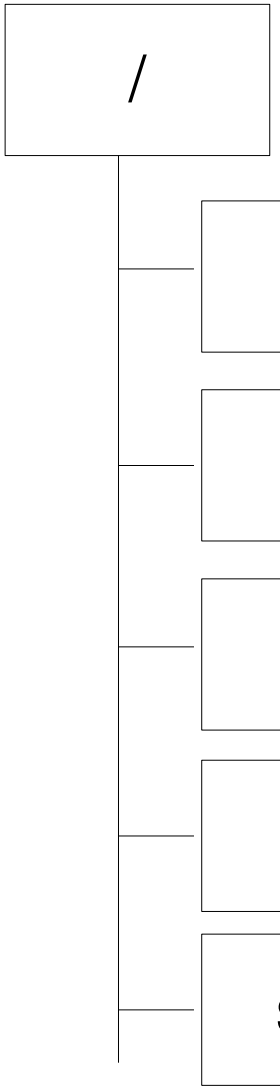
Arborescence



la racine, contient
les répertoires principaux

- **bin** — contient des exécutables essentiels au système, employés par tous les utilisateurs
- **boot** — contient les fichiers permettant à Linux de démarrer
- **dev** — contient le point d'entrée des périphériques
- **etc** — contient les commandes et les fichiers nécessaires à l'administrateur du système (fichiers : passwd, ...)
- **home** — répertoire personnel des utilisateurs

Arborescence



la racine, contient
les répertoires principaux

contient des bibliothèque partagées

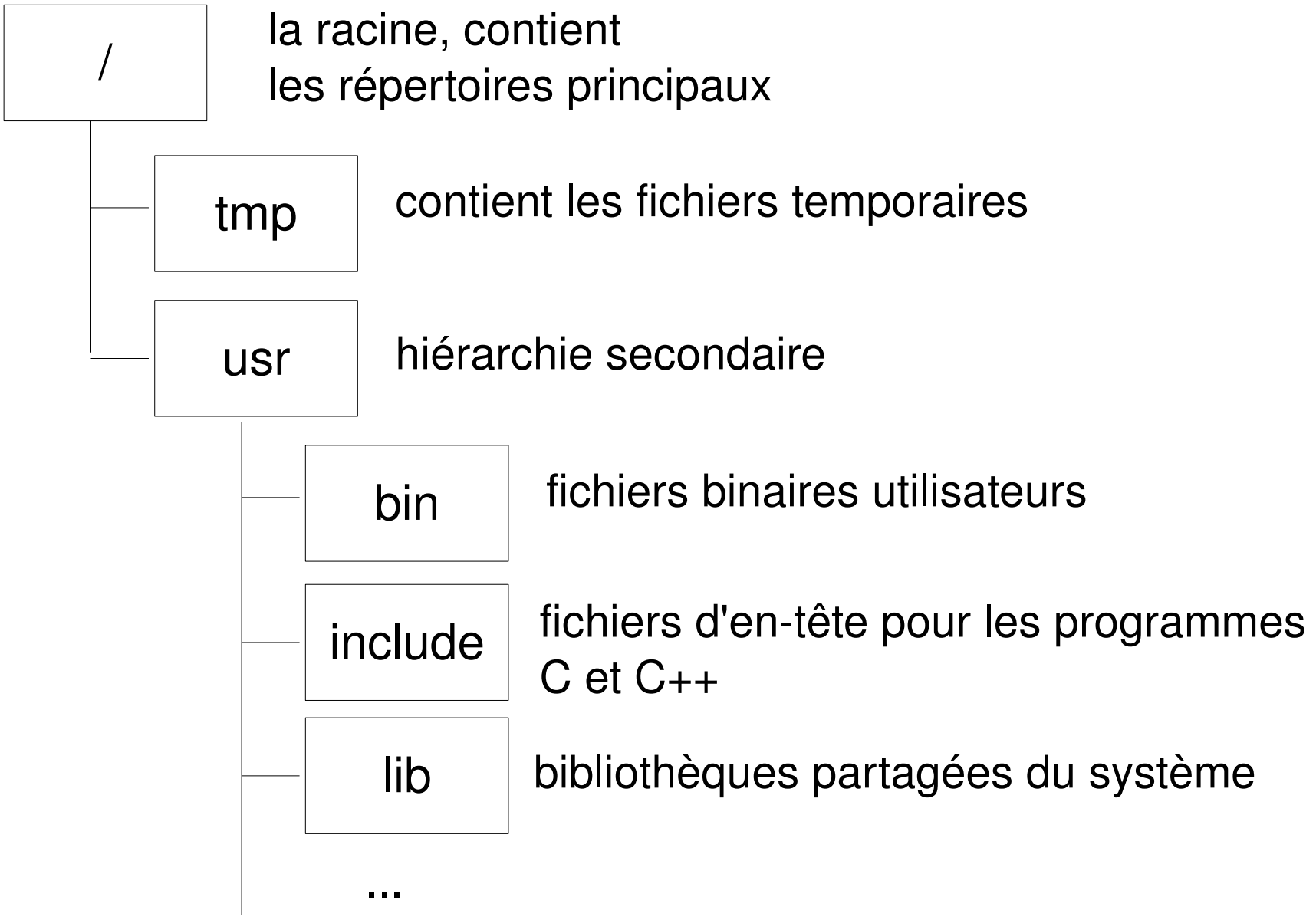
contient le point de montage des partitions temporaires (disquettes, cd-rom, ...)

contient des packages d'application

répertoire de l'administrateur root (super utilisateur)

contient les binaires système essentiels

Arborescence



Commandes pour parcourir l'arborescence

- **cd** : changer de répertoire
  - **cd .** : reste dans le répertoire courant
  - **cd ..** : retourner dans le répertoire père
  - **cd** / : aller à la racine
  - **cd** ~ : aller dans le répertoire de l'utilisateur
  - **cd 'rep'** ou **cd ./'rep'** ou **cd ./'rep'**/ permet de rentrer dans le répertoire 'rep' se trouvant dans le répertoire courant
  - ex: cd /utilisateurs/davigori/Documents/cours/UNIX
- **ls** : liste le contenu d'un répertoire
- **pwd** : permet d'afficher l'arborescence du répertoire actif

différents types de fichiers

- fichiers ordinaires :

contiennent des données ou des programmes.

  - 3 types de fichiers ordinaires :
    - fichiers binaires exécutables, écrits dans le code du processeur de la machine qui ont une structure reconnue par le noyau Linux
    - les fichiers textes qui sont une suite de caractères ASCII structurés en lignes (on peut les ouvrir avec un éditeur texte)
    - autres fichiers binaires qui sont une suite d'octets dont la structure n'est pas reconnue par Linux (mais qui l'est par les logiciels les utilisant)

différents types de fichiers

- répertoire:

  noeuds de la structure arborescente des fichiers. Ils contiennent les identificateurs (i-nodes) des fichiers qu'ils contiennent.

  . : désigne le répertoire courant .. : désigne le répertoire père

- fichiers spéciaux:

  sont liés à un périphérique (/dev/tty01). Corresponde à des programmes (pilotes, drivers) qui gèrent les échanges avec les périphériques

- liens symboliques:

  contiennent le nom d'un autre fichier et permettent des indirections.

lancer un fichier Exécutable

- 2 façons de lancer un fichier exécutable :
  - on se positionne dans son répertoire et on tape le nom du programme précédé de « ./ »

```
davigori@pif:test > ls
prog*
davigori@pif:test > ./prog
le programme tourne!
davigori@pif:test >
```

```
davigori@pif:test > prog
bash: prog: command not found
```

ATTENTION: on doit avoir les droits d'exécution (slide « droits d'accès »)

  - on tape seulement le nom du programme dans un répertoire quelconque. Dans ce cas, le terminal va rechercher le programme dans tous les répertoires décrit par la variable d'environnement « PATH »

PATH

- **PATH**: est une variable d'environnement contenant une liste de répertoire séparée par « : »

```
davigori@pif:test > echo "$PATH"
/utilisateurs/davigori/bin:/usr/lsa/x86_64/bin:/utilisateurs/davigori/bin:
```

C'est dans cette liste de répertoire que le terminal recherche les commandes que l'on écrit au clavier.

- **ajouter le répertoire 'rep' au PATH**: ajouter à un des fichiers de configuration « ~/.bashrc » ou « ~/.bash_profile »

  PATH=$PATH:$HOME/rep

  export PATH

- on peut remarque que les fichiers de configuration commence par un « . », ce sont des fichiers cachés

  la différence entre bashrc et bash_profile est que le premier est appelé lors de l'ouverture d'un terminal et l'autre lors de l'ouverture de la session

Commandes fichiers

- **rm 'fichier'**: supprimer le fichier 'fichier'
- **mv 'fichier' 'nvNom'**: modifie le nom du fichier
- **mv 'fichier' 'rep/'**: déplace le fichier dans le répertoire 'rep'
- **cp 'fichier' 'destination'**: copie le fichier à la destination mentionnée
- **file 'fichier'**: permet de connaître le type du fichier.
- **cat 'fichier'**: affiche le contenu du fichier (le fichier doit être de type texte).
- **more 'fichier'**: affiche le contenu du fichier avec des pauses.

Commandes répertoires

- **mkdir 'rep'**: créer un répertoire VIDE.
- **rmdir 'rep'**: supprimer un répertoire VIDE.
- **rm -r 'rep'**: supprimer le répertoire 'rep' et son contenu (y compris les sous répertoire)
- **mv 'rep' 'nvrep'**: renomme le répertoire 'rep'
- **cp -r 'rep' destination**: copie le répertoire 'rep' et son contenu à la destination spécifiée

création de liens

- il existe 2 types de liens entre fichiers (*Commande : ln 'source' 'dest')*
  - liens matériels (ou physiques)
  - liens symboliques (ou logiques)
- lien matériel: manière de nommer un fichier. Un fichier peut avoir plusieurs noms. Il n'y a pas de notion de nom original. Le fichier est réellement effacé une fois que son dernier nom (lien matériel) est détruit.

=> Un lien matériel doit se trouver sur la partition contenant réellement le fichier

- lien symbolique (utiliser l'option -s) : c'est un petit fichier spécial qui contient un chemin d'accès. Lorsque l'on supprime le lien symbolique (commande : unlink) seul le lien est détruit et nom le fichier.

=> le lien peut se trouver sur une autre partition que le fichier pointé.

```
davigori@pif:UNIX > ls -i *.txt
2688819518 lienPTest.txt*  2688816772 lienTest.txt@  2688819518 test.txt*
```

Droits d'accès

- Linux est un système d'exploitation multi-utilisateur.
- Pour des raisons de sécurité, les fichiers ne peuvent être «lus», «modifiés» ou «exécutés» uniquement par les personnes habilitées.

=> lors de la création d'un fichier, on doit spécifier un certain nombre de permissions

Il y a 3 types d'utilisateur : le propriétaire, le groupe et le « reste du monde »

+ le « super-utilisateur » ou « root » : utilisateur (administrateur) qui est habilité à modifier le système (installation matériel, ajout d'un utilisateur, ...)

su login: permet de changer l'utilisateur d'un terminal

Droits d'accès

ls -l : donne les droits d'accès des fichiers du répertoire

```
davigori@pif:UNIX > ll
total 2188
drwxr-xr-x 2 davigori Midi   4096 jan  8 13:51 images/
-rw-r--r-- 1 davigori Midi 213817 jan  6 18:05 Introduction.odp
-rwxrwxrwx 2 davigori Midi      6 jan  8 13:48 lienPTest.txt*
lrwxrwxrwx 1 davigori Midi      8 jan  8 13:51 lienTest.txt -> test.txt*
-rwxrwxrwx 2 davigori Midi      6 jan  8 13:48 test.txt*
```

- *Type de fichier (un caractère):*
  - *si caractère 'd', c'est un répertoire*
  - *si caractère 'l', c'est un lien symbolique*
  - *si caractère '-', c'est un répertoire*
- *Trois caractères indiquant les droits du propriétaire du fichier :*
  - *si 'r', droit en lecture ('-' sinon)*
  - *si 'w', droit en écriture ('-' sinon)*
  - *si 'x', droit d'exécution ('-' sinon)*
- Trois caractères identique au précédent mais pour le groupe

Droits d'accès

```
davigori@pif:UNIX > ll
total 2188
drwxr-xr-x 2 davigori Midi   4096 jan  8 13:51 images/
-rw-r--r-- 1 davigori Midi 213817 jan  6 18:05 Introduction.odp
-rwxrwxrwx 2 davigori Midi      6 jan  8 13:48 lienPTest.txt*
lrwxrwxrwx 1 davigori Midi      8 jan  8 13:51 lienTest.txt -> test.txt*
-rwxrwxrwx 2 davigori Midi      6 jan  8 13:48 test.txt*
```

- *Trois caractères identique au précédent mais pour le « reste du monde »*
- *Un chiffre pour le nombre de liens*
- *Le nom du propriétaire*
- Le nom du groupe
- La taille du fichier
- La date et l'heure de création
- Le nom du fichier

Droits d'accès

Seuls le propriétaire et le super utilisateur peuvent modifier les droits d'un fichier, avec la commande : ***chmod.***

Il existe 2 modes pour modifier les droits :

- le mode d'accès absolue :
  - les autorisations sont données sous forme d'un nombre octal composé de 3 chiffres.
  - Le premier chiffre correspond aux autorisations du propriétaire, le suivant au groupe et le dernier aux autres utilisateurs.
  - Pour calculer chaque chiffre, on ajoute les valeurs des autorisations en comptant 4 pour read, 2 pour write et 1 pour exécute

```
davigori@pif:UNIX > chmod 750 images
davigori@pif:UNIX > ll
total 2188
drwxr-x--- 2 davigori Midi   4096 jan  8 13:52 images/
```

Droits d'accès

le mode d'accès symbolique : chmod [qui] op permissions

- « qui » désigne celui ou ceux qui recevront ces autorisations : u (propriétéaire), g (groupe), o (les autres), a (les 3, par défaut)
- « op » indique si l'on veut ajouter : « + » ou enlever: « - »
- les permissions sont données par « r », « w », « x »
- ex : chmod ug+wx fichier
- ex : chmod ga+w-x fichier

```
davigori@pif:UNIX > chmod o+rx images
davigori@pif:UNIX > ll
total 2188
drwxr-xr-x 2 davigori Midi   4096 jan  8 13:55 images/
```

Espace Disque

- **du** : affiche l'espace occupé par toute l'arborescence placé sous le répertoire affiché

```
davigori@pif:UNIX > du
636K    ./images
0       ./test
2,9M    .
```

- **df** : affiche les informations du système fichiers placé sous le répertoire affiché

```
davigori@pif:UNIX > df .
Sys. de fich.          Tail. Occ. Disp. %Occ. Monté sur
dumbo:/home1           557G  306G  252G  55% /dumbo-home1
```

Commandes UNIX

Liste de commandes

- commandes usuelles :
  - man, cd, ls, pwd, mkdir, rmdir, mv, cp, top, ps, chmod, ln, kill, su
- manipulation fichiers :
  - cat : visualise ou concatène des fichiers
  - chown : change le propriétaire d'un fichier
  - find : recherche de fichier ou répertoire
  - grep : recherche d'une chaîne de caractère dans un fichier
  - head/tail : affiche le début ou la fin d'un fichier
  - sort : tri les lignes d'un fichier
  - wc : compte le nombre de mots/lignes/caractères d'un fichier
  - more : affiche le contenu d'un fichier
- autres :
  - echo : affiche les messages passés en paramètre

redirections

- avant de lancer un processus le Shell lui associe 3 fichiers :
  - Entrée standard (par défaut clavier); descripteur 0
  - Sortie standard (par défaut écran); descripteur 1
  - Erreur standard (par défaut écran); descripteur 2
- redirection de la sortie standard « > » :
  - ./prog > 'nomfichier'

  la sortie utilisée ne sera plus l'écran mais le fichier 'nomfichier', si il existe il sera écrasé

```
davigori@pif:UNIX > ls -l > fichier
davigori@pif:UNIX > more fichier
total 2324
-rw-r--r-- 1 davigori Midi      0 jan  8 16:09 fichier
drwxr-xr-x 2 davigori Midi   4096 jan  8 15:34 images/
-rw-r--r-- 1 davigori Midi 213817 jan  6 18:05 Introduction.odp
-rwxrwxrwx 2 davigori Midi      6 jan  8 13:48 lienPTest.txt*
```

redirections

- redirection de la sortie standard « >> » :
  - cette fois la sortie standard est ajoutée à la fin du fichier
- redirection du fichier d'erreur « 2> »
- redirection de l'entrée standard « < »
- PIPE : les données de sortie d'une commande peuvent être directement utilisées comme données d'entrée d'une autre commande grâce au symbole « | ».

```
davigori@pif:UNIX > ls
fichier  Introduction.odp  lienTest.txt@  test.txt*  unix.pdf
images/  lienPTest.txt*    test/          unix.odp   unix.ps
davigori@pif:UNIX > ls | grep t
Introduction.odp
lienPTest.txt*
lienTest.txt@
test/
test.txt*
davigori@pif:UNIX >
```

ScriptShell

ScriptShell

On peut regrouper une séquence de commande **Shell** en un programme appelé **script.**

Etapes pour créer un script :

- créer un fichier avec un éditeur de texte en y mettant les commandes souhaitées
- rendre le programme exécutable
- (optionnel) placer le programme dans le HOME/bin

Exemple de script shell

Soit le programme shell copie :

```
#!/bin/bash
#     copie
echo "Nom du programme : $0"
echo "Nb d'arguments : $#"
echo "Source : $1"
echo "Destination : $2"
cp $1 $2
```

#! suivit du type du Shell (première ligne obligatoire)

pour les commentaires

suite de commande

des variables: $

Variables

- une variable est définie par un nom, une suite de caractère.
- Il existe 2 types de variables
  - utilisateur : définis en minuscule (ex: toto)
  - système : en majuscule (PATH, HOME)
- affecter une valeur à une variable:
  - opérateur d'affectation =
  - la commande read

Attention : pas d'espace entre variable, opérateur et valeur

- caractère $
  - lorsque le Shell rencontre $nom, il remplace la variable nom par sa valeur

Variables

- opérateur =

Soit le programme shell :

```
#!/bin/bash

x=salut y=david
echo "y est $y"
echo  "$x$y"
```

résultat :

```
> y est david
> salutdavid
```

Variables

- commande read

Soit le programme shell :

```
#!/bin/bash

echo "veuillez écrire 2 mots"
read a b
echo  "vous avez saisi $a $b"
```

résultat :

```
> veuillez écrire 2 mots
> salut david
> vous avez saisi salut david
```

- Si on utilise la commande read sans paramètre, la ligne lue est enregistrée dans la variable REPLY

Variables

- **Exercices**

  - Ecrire un programme shell *deuxfois* qui affiche le message "Entrez un mot : ", lit le mot saisi par l'utilisateur puis affiche ce mot deux fois sur la même ligne.

  - Ecrire un programme shell *untrois* qui demande à l'utilisateur de saisir une suite de mots constituée d'au moins trois mots et qui affiche sur la même ligne le premier et le troisième mot saisis.

Variables

- chiffre = paramètre de position

=> ne peut pas être utiliser comme nom de variable

- paramètres de position = arguments passés lors de l'appel du script

=> $1 remplacé par la valeur du premier argument

- paramètre spécial $# = nombre d'arguments
- paramètre de position 0 = nom script

```
#!/bin/bash

echo "Nom du programme : $0"
echo "Nb d'arguments : $#"
echo "argument 1 : $1"
```

```
> prog.sh toto
Nom du programme : toto
Nb d'arguments : 1
argument 1 : toto
```

Variables

- commande set

  permet de redéfinir les arguments de position

```
#!/bin/bash

echo "$1 $2"
set tata toto
echo "$1 $2"
```

```
> prog.sh bonjour pierre
bonjour pierre
tata toto
```

Variables

- Exercice

Ecrire un programme shell *cp2fois* prenant trois arguments : le premier désigne le nom du fichier dont on veut copier le contenu dans deux fichiers dont les noms sont passés comme deuxième et troisième arguments. Aucun cas d'erreur ne doit être considéré.

Variables

- suppression d'ambiguïté

Pour éviter les ambiguïtés dans l'interprétation des références de paramètres, on utilise la syntaxe ${paramètre}.

Ex : $x=bon
$x1=jour
echo $x1
=>jour
echo ${x}1
=>bon1

Ex : set un deux trois quatre cinq six sept huit neuf dix onze douze
echo $11
=> un1
echo ${11}
=>onze

Substitution de commandes

- $(cmd) :
  - exécute cmd
  - peut affecter le résultat à une variable ou paramètres de position

Ex : repert=$(pwd) => initialise la variable repert avec /home/davigori

echo "mon repertoire est $repert"

=> mon repertoire est /home/davigori

Substitution de commandes

- Plusieurs commandes peuvent être présentes entre les parenthèses

Ex : pwd ; who

=> /home/davigori

davigori :0 2009-01-05 08:53

```
set $(pwd ; who)
echo "$2: $1     connecté depuis $4"
```

=>davigori: /home/davigori connecté depuis 2009-01-05

- EXERCICE : écrire un programme *jour* qui affiche le jour courant du mois (en chiffre).

Code de retour

- Le code de retour d'un programme shell est le code de retour de la dernière commande qu'il a exécutée.
- **$?** contient le code de retour du dernier programme.
- Si pas d'erreur $? = 0

```
davigori@pif:UNIX > ls vi
ls: vi: Aucun fichier ou répertoire de ce type
[1]   Done                    acroread coursUnix.pdf
[3]-  Done                    ooffice unix.odp
davigori@pif:UNIX > echo $?
2
```

- **exit [n]** : permet d'arrêter le programme en envoyant le code de retour n.
- Si n n'est pas précisé retourne le code de retour de la dernière commande.

choix multiple : case

Syntaxe : case *mot* in
[modèle [| modèle] ...)
suite_de_commandes ;;] ...
esac

- Exemple

```
#!/bin/bash
        #       @(#)    oui
read -p "Entrez votre réponse : " rep
case $rep in
o|O )     echo OUI ;;
n|N )     echo NON ;;
*)        echo Indefini
esac
```

- Exercice

Ecrire un programme shell 4arg qui vérifie que le nombred'arguments passés lors de l'appel du programme est égal à 4 et écrit suivant le cas le message "Correct" ou le message "Erreur".

itération While

Syntaxe : while suite_cmd1
do
suite_cmd2
done

- la commande suite_cmd2 est exécutée tant que suite_cmd1 vaut 0
- Exemple

```
#!/bin/bash

while who | grep davigori
 do
   echo "l\'utilisateur davigori est encore connecte"
   sleep 5
 done
```

- Lecture dans un fichier

```
#!/bin/bash

who > tmp
while read nom reste
do
     echo $nom
done < tmp
rm tmp
```

itération While

- Exercice

  Créer un fichier *personne* :

  david m

  aurelie f

  pierre m

  sophie f

  lucie f

  Créer un programme qui lit ce fichier et affiche uniquement les prénoms féminins

itération For

- première syntaxe :

```
for var
do
        suite_de_commandes
done
```

- var prend successivement la valeur de chaque **paramètre de position** initialisé

programme for_arg

```
#!/bin/bash
for i
do
  echo $i
  echo "Passage a l'argument suivant ..."
done
```

```
> for_arg un deux
un
Passage a l'argument suivant ...
deux
Passage a l'argument suivant ...
>
```

itération For

- deuxième syntaxe :

```
for var in liste_mots
do
        suite_de_commandes
done
```

- var prend successivement la valeur de chaque mots de liste_mots

programme for_liste

```
#!/bin/bash
for a in toto tata
do
  echo $a
done
```

```
> for_liste
toto
tata
>
```

itération For

- Exercice :
  - Ecrire un programme shell **lsrep** ne prenant aucun argument, qui demande à l'utilisateur de saisir une suite de noms de répertoires (read) et qui affiche leur contenu respectif.

IF

- Syntaxe:

```
if commandes1
then
    commandes2
[ elif commandes ; then suite_de_commandes ]...
[ else commandes ]
fi
```

- Si **commandes1** retourne 0 => **commandes2** sera exécuté

  1 => **elif** puis **else**

- exemple: programme rm1

```
-----------------------------------
 #!/bin/bash
 if   rm $1
   then echo $1 a ete supprime
   else echo $1 n\'a pas ete supprime
 fi
-----------------------------------
```

```
> echo salut > toto
> rm1 toto
toto a ete supprime
> rm1 toto
toto n'a pas ete supprime
```

test sur des expressions conditionnelles

- syntaxe: **if** [[expr_cond]]
- Les deux caractères crochets doivent être **collés** + **espace** de part et d'autre de expr_cond
- évaluation de expr_cond si **VRAIE** => retourne 0
- expr_cond peut porter sur différents objets tels que des chaînes de caractères, des fichiers,

expr_cond pour fichiers

- [[-r entrée]] vraie si entrée existe et est accessible en lecture par le processus courant.
- [[-w entrée]] vraie si entrée existe et est accessible en écriture par le processus courant.
- [[-x entrée]] vraie si - le fichier entrée existe et est exécutable par le processus courant ou - le répertoire entrée existe et le processus courant possède la permission de passage.
- [[-f entrée]] vraie si entrée existe et est un fichier ordinaire
- [[- d entrée]] vraie si entrée existe et est un répertoire

expr_cond pour chaîne de caractères

- [[-z ch]] vraie si chaîne est vide
- [[-n ch]] vraie si chaîne non vide
- [[ch1 > ch2]] vraie si ch1 précède ch2
- [[ch1 < ch2]] vraie si ch1 suit ch2
- [[ch == md]] vraie si ch correspond au modèle md
- [[ch != md]] vraie si ch ne correspond pas au modèle md

programme cmp_a

```
#!/bin/bash
if [[ $1 == a* ]]
    then  echo commance par a
   else echo ne commance pas par a
fi
```

```
> cmp_a a tout de suite !
commance par a
> cmp_a salut a bientôt !
ne commance pas par a
```

combinaison d'expr_cond

- ( ) vraie si si les expressions à l'intérieur sont vraies, permet de regrouper des expressions cond
- ! exp vraie si exp est fausse
- (exp1 && exp2) vraie si les deux expr_cond sont vraies.
- (exp1 || exp2) vraie si l'une des deux expr_cond est vraie.

programme notaccesRW

```
----------------------------
#!/bin/bash
   if [[ ! ( -w /etc/at.deny || -r /etc/at.deny ) ]]
   then
            echo OUI
   else
            echo NON
  fi
----------------------------
```

```
> ls -l /etc/at.deny
-rw-r----- 1 root daemon 144 jan 3 2006 /etc/at.deny
> notaccesRW /etc/at.deny
OUI
le fichier n'est ni accessible en lecture ni en
écriture pour l'utilsateur davigori du groupe MIDI
```

Chaînes de caractères

- \ permet d'utiliser les caractères spéciaux comme des caractères normaux (on protège le caractère)

  ex: echo * => affichera *

- " " permet de protéger tous les caractères de chaîne sauf $ \ ` "

  ex : echo "$toto*" => affichera la valeur de toto suivit du caractère *

- ' ' permet de protéger tous les caractères

  ex echo '$toto' => affichera $toto

Chaînes de caractères

- longueur d'une chaîne de caractères

  Syntaxe : ${#paramètre}

ex :

```
davigori@pif:documents >  echo ${#PWD}
31
davigori@pif:documents > pwd
/utilisateurs/venkiah/documents
```

```
davigori@pif:documents > set "au revoir"
davigori@pif:documents > echo ${#1}
9
```

modificateur de chaîne

suppression de la plus courte sous chaîne à gauche

Syntaxe : ${paramètre#modèle}

ex : set "12a34aB" ; echo ${1#*a} => 34aB

suppression de la plus longue sous chaîne à gauche

Syntaxe : ${paramètre##modèle}

ex : set "12a34aB" ; echo ${1##*a} => B

modificateur de chaîne

- suppression de la plus courte sous chaîne à droite

  Syntaxe : ${paramètre%modèle}

  ex : set "12a34aB" ; echo ${1%a*} => 12a34

- suppression de la plus longue sous chaîne à droite

  Syntaxe : ${paramètre%%modèle}

  ex : set "12a34aB" ; echo ${1%%a*} => 12

modificateur de chaîne

Syntaxe : ${paramètre:ind}

Extraction de la sous-chaîne à partir de l'indice ind

ex : set "12a34aB" ; echo ${1:3} => 34aB

Syntaxe : ${paramètre:ind:nb}

Extraction de la sous-chaîne denb caractère à partir de l'indice ind

ex : set "12a34aB" ; echo ${1:3:2} => 34

Expressions Arithmétiques

- syntaxe: ((expr_arith))
- on peut utiliser les expressions arithmétiques du C
- déclaration variable entière: declare –i b (facultatif)

programme compteur

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

declare -i i=0
while (( i<$1))
do
  echo $1
  (( $1 = $1 +1 ))
 done
--------------------
```

- Exercice : Créer un programme demain, qui affichera la date de demain.

FOR pour Expressions Arithmétiques

- Syntaxe:

```
for (( expr_arith1 ; expr_arith2 ; expr_arith3 ))
do
        suite_cmd
done
```

- Exemple:

```
for (( x=0 ; x<5 ; x++ ))
do
      echo $(( x*2 ))
done
```

Tableaux

- **déclaration:** declare -a nomtab
  déclaration facultative, il suffit d'initialiser un de ses éléments
- **initialisation:** nomtab=(val0 val1 ...)
  nomtab=([indice] = val ...)
  read -a nomtab
- **affectation**: nomtab[3]=val
- **accès:** ${nomtab[indice]}
- **longueur d'un élément:** ${#nomtab[indice]}
- **nombre d'éléments:** ${#nomtab[*]}

Tableaux

- Exercice : Ecrire un programme shell distrib qui crée un paquet de 5 cartes différentes tirées au hasard parmi un jeu de 32 cartes et affiche le contenu du paquet.

PS: on utilisera la variable RANDOM pour obtenir une valeur aléatoire

Fonctions Shell

- Syntaxe:

```
function nom_fct
{
    suite_de_commandes
}
```

- pour appeler une fonction, il suffit de mentionner son nom

```
function f0
{
    echo Bonjour tout le monde !
}

> f0
Bonjour tout le monde !
>
```

fonctions: arguments

- utilisation des paramètres de position

```
#!/bin/bash
# prog
function f
{
echo " --- Dans f : \$0 : $0"
echo " --- Dans f : \$# : $#"
echo " --- Dans f : \$1 : $1"
}
echo " --- Dans prog : \$0 : $0"
echo " --- Dans prog : \$1 : $1"
echo " --- Dans prog : \$2 : $2"

f un deux
```

```
> prog un deux trois
prog
un
deux
f
2
un
>
```

MAKEFILE

Définition

- Makefiles : fichiers, appelés **makefile** ou **Makefile**, utilisés par le programme **make** (se trouvant dans usr/bin).
  => fichier makefile non exécutable !!
- Fonction: exécuter un ensemble d'actions pour compilation d'un projet, archivage de documents, la mise à jour de site, ...

Nous présenterons le fonctionnement de makefile au travers de la compilation d'un petit projet en C.

Règles

- Un Makefile est un fichier constitué de plusieurs règles de la forme :

```
cible: dépendances
        commandes
```

- Chaque commande est précédée d'une **tabulation**.

makefile

```
hello: hello.o main.o
        gcc -o hello hello.o main.o
hello.o: hello.c
        gcc -o hello.o -c hello.c -W -Wall -ansi -pedantic
main.o: main.c hello.h
        gcc -o main.o -c main.c -W -Wall -ansi -pedantic
```

Exécution du make

- Lors appel au makefile via la commande make:
  - la première règle rencontrée, ou la règle dont le nom est spécifié, est évaluée.
- L'évaluation d'une règle se fait en plusieurs étapes :
  - Analyse des dépendances.
    Si une dépendance est la cible d'une autre règle => évaluation de cette règle.
  - Evaluation d'une règle :
    - Si la cible ne correspond pas à un fichier
    - Ou Si fichier est plus récent que la règle
      => exécution des commandes

Exécution du make

makefile

```
hello: hello.o main.o
	gcc -o hello hello.o main.o
hello.o: hello.c
	gcc -o hello.o -c hello.c -W -Wall -ansi -pedantic
main.o: main.c hello.h
	gcc -o main.o -c main.c -W -Wall -ansi -pedantic
```

- But : créer le fichier hello
- Première dépendance hello.o est la cible d'une des règles
  => évaluation de cette règle
- Aucune dépendance de hello.o n'est une règle
  => pas d'autre règle à évaluer pour compléter celle-ci.
- Exécute des commandes de hello.o ?
  - Si le fichier hello.c est plus récent que le fichier hello.o, Oui
  - Sinon, Non
- Evaluation de la règle hello.o est terminée
  => Les autres dépendances de hello sont examinées.

Règles complémentaires

- Un makefile doit permettre de forcer la regénération intégrale du projet
  => doit contenir les 3 règles suivantes :
  - all : généralement la première du fichier, elle regroupe dans ces dépendances l'ensemble des exécutables à produire.
  - clean : elle permet de supprimer tout les fichiers intermédiaires.
  - mrproper : suppression des exécutables à produire + appel à clean

Règles complémentaires

makefile

```
all: hello

hello: hello.o main.o
    gcc -o hello hello.o main.o
hello.o: hello.c
    gcc -o hello.o -c hello.c -W -Wall -ansi -pedantic
main.o: main.c hello.h
    gcc -o main.o -c main.c -W -Wall -ansi -pedantic

clean:
    rm -rf *.o

mrproper: clean
    rm -rf hello
```

Variables

- Déclaration : NOM=VALEUR
- utilisation : $(NOM)
- Intérêt : permet de faciliter les évolutions.
  Ex : Déclarer le nom du compilateur dans une variable permet, si on change de compilateur au cours du projet, de ne pas à devoir modifier toutes les commandes. Seule la variable est à modifier.

Variables

- Liste des variables usuelles pour la compilation d'un projet en C :
  - CC : nom du compilateur
  - CFLAGS : options de compilation
  - LDFLAGS : options de l'édition de liens.
  - EXEC : nom des exécutables à générer.

Variables

makefile

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

hello: hello.o main.o
	$(CC) -o hello hello.o main.o $(LDFLAGS)
hello.o: hello.c
	$(CC) -o hello.o -c hello.c $(CFLAGS)
main.o: main.c hello.h
	$(CC) -o main.o -c main.c $(CFLAGS)

clean:
	rm -rf *.o

mrproper: clean
	rm -rf $(EXEC)
```

Variables internes

$@ : nom de la cible

$< : nom de la première dépendance

$^ : liste des dépendances

$? : liste des dépendances plus récentes que la cible

$* : nom du fichier sans suffixe

Variables internes

makefile

```
CC=gcc
CFLAGS=-W -Wall -ansi -pedantic
LDFLAGS=
EXEC=hello

all: $(EXEC)

hello: hello.o main.o
    $(CC) -o $@ $^ $(LDFLAGS)

hello.o: hello.c
    $(CC) -o $@ $< $(CFLAGS)

main.o: main.c hello.h
    $(CC) -o $@ $< $(CFLAGS)

clean:
    rm -rf *.o

mrproper: clean
    rm -rf $(EXEC)
```

Règle d'inférence

- Makefile permet également de créer des règles génériques qui se verront appelées par défaut. (par exemple construire un .o à partir d'un .c)
- Syntaxe :

```
%.o: %.c
   commandes
```

- Exemple :

```
%.o: %.c
   $(CC) -o $@ -c $< $(CFLAGS)
```

=> doit utiliser les variables internes

($@ : nom de la cible ; $< : première dépendance)

Génération liste objets

- Génération automatique de la liste d'objets à partir de la liste des fichiers sources.
- On a besoin de définir 2 variables :
  - SRC qui contient la liste des fichiers sources du projet.
  - OBJ pour la liste des fichiers objets.
- Syntaxe : OBJ= $(SRC:.c=.o)
  récupère la liste contenue dans SRC et remplace .c par .o

```
SRC= hello.c main.c
OBJ= $(SRC:.c=.o)
```

Génération liste SRC

- wildcard *.c : permet de récupérer tous les fichiers d'extension .c du répertoire courant

```
SRC= $(wildcard *.c)
OBJ= $(SRC:.c=.o)
```

Commandes silencieuses

- @ : permet de désactiver l'affiche des messages des commandes précédées de ce caractère.

```
clean:
   @rm -rf *.o
```

Makefile conditionnel

- Syntaxe :

```
ifeq($(VAR), valeur)
    cmd
else
    cme
endif
```

- Exemple :

makefile

```
DEBUG=yes
ifeq ($(DEBUG),yes)
   CFLAGS=-W -Wall -ansi -pedantic -g
   LDFLAGS=
else
   CFLAGS=-W -Wall -ansi -pedantic
   LDFLAGS=
endif
```

Options de la Commade make

- make -n :
  Ne pas lancer la compilation, mais juste dire ce que l'on ferait.
- make -k :
  Continuer même s'il y a des erreurs.
- make -j 5 :
  On peut lancer jusqu'à 5 commandes à la fois (pertinent sur les machines multi-processeurs).
- make -f Makefile.linux all :
  permet de préciser le nom du fichier makefile

Éditeur Texte : VIM

VIM

- Vim est un éditeur dit modal
  => 2 modes d'exécution :
  - Mode entrée : on peut taper du texte comme on le ferait sur un éditeur quelconque.
  - Mode commande : permet d'exécuter des commandes de manipulation du texte: déplacement, recherche, copie, sauvegarde.
-

Changement de mode

- mode commande : [Echap]
- mode entrée :
  - i : insérer du texte à l'endroit du curseur (ou [Inser])
  - a : insérer du texte après le curseur
  - I : insérer du texte en début de ligne
  - A : insérer du texte en fin de ligne
  - o : commencer une nvelle ligne après le curseur
  - O : commencer une nvelle ligne avant le curseur

déplacement

- Utilisation des flèches
- En mode commande :
  - w : avance d'un mot
  - b : recule d'un mot
  - $: avance jusqu'à la fin de la ligne
- On peut utiliser un argument numérique devant une commande. Ex : 10w, avancera de 10 mots

Copier, couper, coller

- yy : copie la ligne courante dans le tampon (5yy : copie 5 lignes)
- p : colle le contenu du tampon après la ligne courante
- P : colle le contenu du tampon avant la ligne
- D : détruite le reste de la ligne depuis le curseur
- dd : détruit entièrement la ligne
- dw : efface le mot à droite du curseur
- db : efface le mot à gauche du curseur

Charger / Sauver

- :q : quitte vim sans enregistrer
- :q! : force à quitter vim sans enregistrer
- :w : sauve le fichier sous le nom courant
- :wq : quitte en sauvant
- :r fichier_à_inserer : insère le fichier à l'endroit du curseur
- :e fichier_à_editer : charge un nouveau fichier

VIM

- Pour plus d'info, utiliser le tutoriel :
  - vimtutor