Moustapha DIABY

moustapha.diaby@adoptimax.com

http://www.diabymoustapha.adoptimax.com

ESATIC

Systèmes d'exploitations des ordinateurs

A. Généralités
B. Unix
    1. Généralités
    2. Système de Gestion des Fichiers
    3. Les processus
    4. Langage de commandes

Système d'exploitation des ordinateurs

Module Système – Semestre 1

Esatic, UP informatique

Année 2014-2015

Moustapha Diaby
(moustapha.diaby@adoptimax.com – http://www.diabymoustapha.adoptimax.com)

Partie A

Présentation générale des systèmes d'exploitations

Département Informatique

Ordinateur

Un ordinateur contient :

- un (ou des) **processeurs** ..... calcul
- des horloges ..... temporisation
- des terminaux ..... interaction ordinateur/utilisateur
- une **mémoire** principale ..... stockage non-durable
- une mémoire secondaire (disques, flash, etc.) .... stockage durable
- des interfaces de connexion à des réseaux
- des périphériques d'entrées/sorties

Il sert **uniquement** à :

1. Faire des calculs
2. Stocker les résultats de ces calculs

Logiciels

Un logiciel est une suite prédéfinie d'instructions (un programme) pour effectuer un *calcul*.

Il existe deux catégories de logiciels :

① **les programmes systèmes** . . . . . fonctionnement des ordinateurs

② **les programmes d'applications** . . . . . résolution des problèmes utilisateurs

système d'exploitation = programme système fondamental.

Il contrôle les ressources de l'ordinateur et fournit la base sur laquelle se construisent les programmes d'applications.

Le système d'exploitation fonctionne sous deux modes différents :

☞ mode **noyau** (ou **superviseur**) . . . . . contrôle total de l'ordinateur

☞ mode **utilisateur** . . . . . contrôle restreint

Deux fonctions

Un système d'exploitation doit remplir deux fonctions :

① **MACHINE VIRTUELLE**
masquer les élements fastidieux liés au matériel, comme la gestion de la mémoire, la gestion des périphériques (déplacement de la tête de lecture d'un disque, etc.), les horloges, etc.

② **GESTIONNAIRE DE RESSOURCES**
ordonnancer et contrôler l'allocation des ressources (processeurs, mémoire, entrées/sorties, etc.) entre les différents programmes qui y font appel.

Le rôle de gestionnaire des ressources est *crucial* pour les systèmes d'exploitations manipulant plusieurs tâches en même temps (multi-tâches).

Classes de systèmes d'exploitation

Il existe différentes classes de systèmes d'exploitations en fonction :

- ☞ des services rendus
- ☞ de leur architecture logique
- ☞ de l'architecture matérielle qui les supporte
- ☞ de leur couplage

Services rendus

- **MONO/MULTI TÂCHES** :
  capacité du système à pouvoir (ou pas) exécuter plusieurs processus simultanément

- **MONO/MULTI UTILISATEURS** :
  capacité à pouvoir (ou pas) gérer un panel d'utilisateurs utilisant simultanément les mêmes ressources matérielles

Architecture des systèmes

- **SYSTÈMES CENTRALISÉS** :
  - l'ensemble du système est entièrement présent sur la machine
  - les machines éventuellement reliées sont vues comme des entités étrangères disposant elles aussi d'un système centralisé
  - ne gère que les ressources de la machine sur laquelle il est présent
- **SYSTÈMES RÉPARTIES OU DISTRIBUÉS** :
  - les différentes abstractions du système sont réparties sur un ensemble (domaine) de machines (site)
  - le système apparaît aux yeux de l'utilisateur comme une machine unique (virtuelle) monoprocesseur
  - l'utilisateur ne se soucie pas de la localisation des ressources
  - exploitation des capacités de parallèlisme d'un domaine
  - assez résistant aux pannes

Architecture matérielle

- **MONOPROCESSEUR** : *temps partagé ou multiprogrammation*
  Il a fallu développer un mécanisme de gestion des processus pour offrir un pseudo parallélisme à l'utilisateur. En fait c'est une communication rapide entre processus qui permet de donner l'illusion du parallélisme.
- **MULTIPROCESSEUR** : *parallélisme*
  Différentes variétés
  - **SIMD (Single Instruction Multiple Data)** : tous les processeurs exécutent les même instructions mais sur des données différentes
  - **MIMD (Multiple Instruction Multiple Data)** : chaque processeur est complètement indépendant et exécute des instructions sur des données différentes
  - **Pipeline** : les différentes unités d'exécution sont mises en chaîne et font chacune une partie du traitement à effectuer.

Partie B

UNIX

Département Informatique

Cours n° B.1

Généralités sur UNIX

Philosophie d'UNIX

- ☞ Le code source est (souvent) disponible et facile à lire
- ☞ L'interface utilisateur est simple (pas forcément très conviviale mais simple)
- ☞ **Il n'y a qu'un petit nombre de primitives mais les combinaisons sont très nombreuses**
- ☞ Toutes les interfaces avec les périphériques sont unifiées (via le système de gestion des fichiers)
- ☞ Le système est *indépendant* de l'architecture matérielle

Caractéristiques d'UNIX

UNIX est un système d'exploitation :

- ☞ multi-utilisateurs
- ☞ multi-tâches
- ☞ qui possède un système de gestion des fichiers à arborescence unique, même avec plusieurs périphériques de stockage
- ☞ dont les entrées/sorties et la communication inter-processus sont compatibles avec la notion de fichier (interface de manipulation unique)

Terminologie

Une architecture en couche

Le fonctionnement d'UNIX est basé sur une architecture logicielle en couche :

- ☞ Programmes utilisateurs
- ☞ Programmes systèmes
- ☞ Noyau du système
- ☞ Matériel (*hardware*)

Interpréteur de commandes

Un programme particulier permet aux utilisateurs de communiquer avec le système via un langage de commandes :

l'interpréteur de commandes (*shell*)

L'algorithme de ce programme utilisateur indispensable est le suivant :

① Afficher un message d'invite (*prompt*)

② Attendre la validation d'une ligne de commandes (touche 

)

③ Interpréter (traduire et exécuter) les commandes données

④ Retourner en ①

Syntaxe générale des commandes UNIX

Les différents langages de commandes (*shells*) utilisent tous la même syntaxe générale pour la description d'une commande :

```
commande [options...] [arguments...]
```

Une commande peut (cela n'est pas obligatoire) être suivie

- ☞ d'*options* qui précisent le mode de fonctionnement de la commande, une façon particulière de fonctionner

  ➥ COMMENT

- ☞ de *paramètres* ou *arguments* qui permettent de spécifier des éléments que la commande doit prendre en compte

  ➥ QUOI

Une ligne de commande peut comporter plusieurs commandes si elles sont séparées les unes des autres par le caractère point-virgule « ; »

La commande `wc` permet de compter des caractères, des lignes et des mots.

```
$ wc -l fichier
3 fichier
```

L'**option** `-l` indique à la commande `wc` de ne compter que les lignes de l'**argument** `fichier`

Les options sont *souvent* précédées par un signe moins «-»

La commande `man` permet d'obtenir le manuel d'utilisation d'une commande.

```
$ man wc
```

L'**argument** `wc` permet de préciser à la commande `man` de donner la documentation de `wc`.

```
$ man 1 wc
```

L'**option** `1` permet de préciser à la commande `man` de n'aller chercher la documentation de `wc` que dans la section 1 du manuel.

Erreurs

L'interpréteur de commandes ne peut exécuter une ligne de commandes que si elle est exécutable (*valide* syntaxiquement ET sémantiquement).

S'il ne peut pas exécuter une ligne il retournera une erreur. Les cas d'erreurs les plus fréquents sont :

- La commande n'existe pas
- Vous n'avez pas le droit d'exécuter la commande
- Les options de la commande sont erronées
- Les arguments de la commande sont erronés

Dans les deux derniers cas d'erreurs l'utilisation du manuel en ligne (via `man`) permettra d'obtenir plus de détails sur le fonctionnement de la commande.

Cours n° B.2

Système de gestion des fichiers

Philosophie

- Sous UNIX : **TOUT EST FICHIER** [a]
- Du point de vue interne (noyau) les fichiers ont tous la même structure
- Du point de vue utilisateur il existe différents types de fichiers :
  - ordinaires (ou réguliers)
  - catalogues (ou répertoires)
  - liens symboliques
  - spéciaux
  - tubes
  - sockets
- Il y a une représentation hiérarchique du stockage des fichiers

a. ou presque...

Structuration

Pour le noyau un fichier est une suite non-structurée d'octets
(*byte stream*)

Il n'y a pas de structuration directe au niveau du noyau mais il peut y en avoir une au niveau des applications.

Par exemple les fichiers textes :

- ☞ Ce sont des fichiers constitués d'une séquence de lignes.
- ☞ Une ligne est une suite de caractères terminée par le caractère de passage à la ligne.
- ☞ Chaque caractère est représenté par un octet suivant le code ASCII.
- ☞ Le caractère de passage à la ligne est le caractère de code 10 «`\n`».

➠ Cette structuration n'est qu'une convention utilisée par des programmes et non par le noyau du système d'exploitation.

Inodes (1)

- ☞ Les caractéristiques d'un fichier sont stockées dans un bloc de données
- ☞ Le système gère une table avec l'adresse de tous les blocs disponibles
- ☞ Un bloc est repéré par son numéro dans cette table : **son inode**

Le noyau du système gère cette table (c'est-à-dire l'ensemble des inodes)
➠ Notion d'un **SYSTÈME DE FICHIERS**

Il y a un système de fichiers par zone de stockage matérielle (partition d'un disque dur par exemple)

➠ **Un fichier est repéré de manière unique par :**

① **le système de fichier auquel il est attaché**

② **son inode**

Inodes (2)

Fichiers réguliers (vue utilisateur)

Un fichier régulier est un fichier qui n'est ni un catalogue, ni un lien, ni un fichier spécial, ni un tube, ni une socket.

Quelques commandes utilisables sur des fichiers réguliers :

`stat`	permet d'afficher les caractéristiques de base de fichier(s)
`cat`	permet d'afficher le contenu de fichier(s)
`touch`	permet de modifier les caractéristiques de dates de fichier(s). Cette commande permet également de créer un (ou des) fichier(s) vide(s)
`file`	permet de déterminer la convention de structure que respecte le contenu du fichier (donc son *type applicatif*)

Fichiers réguliers (vue noyau)

Fichiers catalogues (vue utilisateur)

Un catalogue (*directory*), ou répertoire, est un fichier qui contient une liste de fichiers.

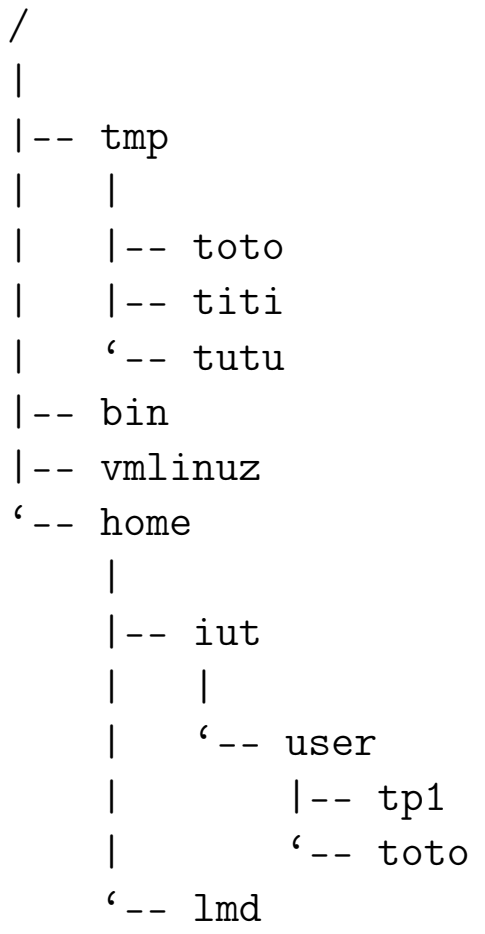
Un répertoire *contient* d'autres fichiers et peut donc contenir un ou des répertoires.

➠ Notion de hiérarchie (ou d'arbre)

Toutes les versions d'UNIX ont une hiérarchie unique, dont le sommet est nommé «/» (*slash*).

Ce répertoire de base est la racine (*root*) de l'arbre hiérarchique.

Ce répertoire a toujours comme inode la valeur 2


/
|
|-- tmp
| |
| |-- toto
| |-- titi
| `-- tutu
|-- bin
|-- vmlinuz
`-- home
    |
    |-- iut
    | |
    | `-- user
    | |-- tp1
    | `-- toto
    `-- lmd

Chemins

Pour identifier un fichier dans la hiérarchie on a besoin :

① du chemin jusqu'au répertoire dans lequel il est stocké

② de son nom

chemin	=	point de départ	+	liste des répertoires à traverser pour arriver au répertoire destination

La liste des répertoires est composée de répertoires séparés les uns des autres par le caractère «/»

☞ si le point de départ est la racine ➠ **chemin absolu**

☞ sinon ➠ **chemin relatif à un autre répertoire**

chemin absolu ➙ `/home/iut/user/toto`

chemin absolu ➟ `/home/iut/user/toto`

chemins relatifs à `/tmp` ➟ `../home/iut/user/toto`

Tous les répertoires contiennent obligatoirement dans leur liste deux fichiers :

- « . » qui est un synonyme pour le répertoire lui-même
- « . . » qui est un synonyme pour le répertoire qui le contient (son père)

Les fichiers dont le nom commence par un point « . » sont appelés *fichiers cachés* (par exemple par défaut la commande `ls` ne les montre pas).

chemin absolu → `/home/etudiants/logina/toto`

chemins relatifs à `/tmp` → `../home/iut/user/toto`

ou `../bin/../home/iut/user/toto`

... etc ...

Quelques commandes utilisables à propos des répertoires :

`pwd`	permet d'obtenir le nom absolu du répertoire de travail courant
`cd`	permet de changer le répertoire de travail courant
`ls`	permet d'obtenir la liste des fichiers contenus dans un répertoire. Il existe de très nombreuses options parmi lesquelles : `-a` permet de voir les fichiers cachés `-i` permet de voir les inodes associées `-l` permet d'avoir les informations pour chaque fichier sur une ligne
`mkdir`	permet de créer un répertoire
`rmdir`	permet de supprimer un répertoire vide

Sans argument `ls` liste le contenu du répertoire de travail courant

`ls -l`

Type	**Caractères**
fichier régulier	`-`
répertoire	`d`
lien (symbolique)	`l`
tube	`p`
socket	`s`
spécial	`c` ou `b`

Fichiers catalogues (vue noyau)

Catalogues	=	Fichier comme un autre
	=	Caractéristiques + Contenu
Contenu	=	Liste de fichiers
	=	Ensemble de couples : (nom, inode)

➥ *entrée* d'un répertoire (un des couples de la liste)

Noms des fichiers (vue noyau)

Le nom de fichier n'est rien d'autre qu'une entrée dans un répertoire. Un nom est donc un **lien physique** (*hard link*) vers un fichier.

Plusieurs entrées de répertoires peuvent utiliser la même inode.

➠ **Un même fichier peut avoir plusieurs noms.**

Manipulation du système de fichiers

Les commandes de manipulation des fichiers à l'intérieur d'un système de fichiers ont souvent la syntaxe suivante :

`commande` <*emplacement_source*> <*emplacement_destination*>

`cp` ➟ permet de copier un fichier :

① création (ou modification) d'un fichier en dupliquant le contenu d'un autre fichier

② création (ou modification) d'une entrée dans un répertoire

`mv` ➟ permet de déplacer un fichier (donc aussi de le renommer) :

① suppression d'une entrée dans un répertoire

② création (ou modification) d'une nouvelle entrée dans un répertoire

Liens symboliques

Un **lien symbolique** (*soft link*) est un fichier (de type lien) qui contient le chemin et le nom d'un autre fichier.

Les accès à un lien ne sont rien d'autre que des redirections vers un autre fichier : les commandes qui manipulent un fichier lien manipule en fait le fichier dont le chemin est stocké dans le lien.

➠ Un lien est donc un raccourci (ou un alias) vers un autre fichier

Le contenu du fichier doit être un chemin

- ☞ soit absolu
- ☞ soit relatif. Dans ce cas le chemin doit être **valide depuis le répertoire dans lequel se trouve le fichier**.

Abus de langage

Par abus de langage on a donc 2 notions différentes :

- ☞ les **liens physiques** ou **hard**, plusieurs entrées de répertoires utilisant la même inode ..... (fichiers de type régulier)

- ☞ les **liens symboliques** ou **soft**, plusieurs inodes différentes dont le contenu désigne un même fichier régulier .... (fichiers de type lien)

La commande `ln` permet de créer des liens :

- ☞ sans option elle permet de créer des liens physiques
- ☞ avec l'option `-s` elle permet de créer des liens symboliques

```
{epicea08-beaufils-/tmp} pwd
/tmp


{epicea08-beaufils-/tmp} ls -ali
total 12
 24801 drwxrwxrwt    5 root     root 4096 Nov 27 00:58 .
     2 drwxr-xr-x   21 root     root 4096 Feb 19  2000 ..
 24802 -rw-rw-r--    3 beaufils ens    13 Nov 27 00:57 toto


{epicea08-beaufils-/tmp} mkdir tata
{epicea08-beaufils-/tmp} ln toto tutu
{epicea08-beaufils-/tmp} ln -s tutu titi


{epicea08-beaufils-/tmp} ls -li
total 12
24804 drwxrwxr-x    2 beaufils ens     4096 Nov 27 00:59 tata
24803 lrwxrwxrwx    1 beaufils ens        4 Nov 27 00:58 titi -> tutu
28402 -rw-rw-r--    3 beaufils ens       13 Nov 27 00:57 toto
24802 -rw-rw-r--    3 beaufils ens       13 Nov 27 00:57 tutu
```

Hiérarchie standard UNIX

`/bin`	Commandes utilisateurs essentielles
`/dev`	Fichiers de périphériques
`/etc`	Fichiers de configuration spécifique à la machine
`/home`	Répertoires des utilisateurs
`/lib`	Librairies partagées
`/sbin`	Commandes d'administration essentielles
`/tmp`	Fichiers temporaires
`/usr`	Seconde hiérarchie
`/var`	Données variables
`/usr/X11R6`	X Window System, version 11 release 6
`/usr/bin`	La plupart des commandes utilisateurs
`/usr/include`	Fichier d'entêtes pour les programmes C
`/usr/lib`	Librairies
`/usr/local`	Hiérarchie locale
`/usr/sbin`	Commandes d'administrations non-vitales
`/usr/share`	Données indépendantes de l'architecture
`/usr/src`	Code source

Plus de détails sur l'effort de standardisation : `http://www.pathname.com/fhs/`

Utilisateurs/Groupes

UNIX est un système multi-utilisateurs. Les utilisateurs y sont rassemblés par groupe. Chaque utilisateur est donc identifié par le système par :

① son *login* ..... au niveau noyau c'est un numéro unique : l'`uid`

② son *groupe* ..... au niveau noyau c'est un numéro unique : le `gid`

Le système gère la correspondance entre identifiant symbolique et numérique via des fichiers textes :

☞ login et `uid` via le fichier `/etc/passwd`

☞ groupe et `gid` via le fichier `/etc/group`

Un utilisateur peut appartenir à plusieurs groupes, mais possède un groupe principal (spécifié dans le fichier `/etc/passwd`) dans lequel il est enregistré lors de chaque connexion.

Droits d'accès

Chaque fichier :

- ☞ appartient à un utilisateur (son *propriétaire*) et à un groupe.
- ☞ posssède des droits d'utilisation applicables :
  1. à son propriétaire
  2. aux utilisateurs appartenant à son groupe
  3. aux utilisateurs n'appartenant pas à son groupe

Pour chacune de ces trois catégories, il existe trois types de droits :

1. **lecture** : autorise la lecture du contenu du fichier
2. **écriture** : autorise la modification du contenu du fichier
3. **exécution/franchissement** :
   - autorise l'exécution d'un fichier régulier,
   - permet de traverser un répertoire

➠ Pour manipuler le système de fichier (copie, déplacement, etc.) un utilisateur doit avoir les droits correspondants sur les fichiers qu'il veut manipuler

L'option `-l` de la commande `ls` permet de voir les droits d'accès d'un fichier. Pour chacun des trois cas d'applicabilité les droits sont affichés par une chaîne de caractère avec la représentation suivante :

- `r` : l'accès en lecture est autorisé
- `w` : l'accès en écriture est autorisé
- `x` : l'accès en exécution/franchissement est autorisé
- `-` : à la place de `r`, `w` ou `x` signifie que l'accès correspondant n'est pas attribué.

chmod

Le mode d'utilisation d'un fichier est l'ensemble de ses droits d'accès.

La commande `chmod` permet au propriétaire d'un fichier de modifier son mode d'utilisation.

La syntaxe de `chmod` est la suivante :

`chmod` <*`mode`*> <*`fichiers`*>

Le mode peut être précisé de deux manières :

- ☞ via la spécification des modifications à effectuer sur le mode courant :
  ➟ forme **symbolique**.
- ☞ via la spécification complète du nouveau mode :
  ➟ forme **numérique octale** (base 8)

`chmod` (forme symbolique)

Les modifications à effectuer sur le mode courant sont spécifiées par un code dont la syntaxe est :

<*personne*><*action*><*accès*>

<*personne*>	
u	propriétaire
g	groupe
o	autres
a	tous

<*action*>	
+	ajouter
-	enlever
=	initialiser

<*accès*>	
r	lecture
w	écriture
x	exécution/franchissement

- ☞ Il ne peut y avoir qu'une action par code
- ☞ Plusieurs modifications peuvent être spécifiées si elles sont séparées les unes des autres par des virgules «,».

```
{epicea08-beaufils-~/tmp} ls -l
total 4
drwxr-x---    2 beaufils ens          4096 Oct 20 14:50 titi
-rw-r--r-x    1 beaufils ens             0 Oct 20 14:51 toto


{epicea08-beaufils-~/tmp} chmod u+x toto
{epicea08-beaufils-~/tmp} ls -l
total 4
drwxr-x---    2 beaufils ens          4096 Oct 20 14:50 titi
-rwxr--r-x    1 beaufils ens             0 Oct 20 14:51 toto


{epicea08-beaufils-~/tmp} chmod og-r toto
{epicea08-beaufils-~/tmp} ls -l
total 4
drwxr-x---    2 beaufils ens          4096 Oct 20 14:50 titi
-rwx-----x    1 beaufils ens             0 Oct 20 14:51 toto


{epicea08-beaufils-~/tmp} chmod a=wx titi
{epicea08-beaufils-~/tmp} ls -l
total 4
d-wx-wx-wx    2 beaufils ens          4096 Oct 20 14:50 titi
-rwx-----x    1 beaufils ens             0 Oct 20 14:51 toto
```

`chmod` (forme numérique octale)

Les différentes combinaisons de droits d'accès peuvent être représentées par :

symbolique	binaire	octal
`---`	000	0
`--x`	001	1
`-w-`	010	2
`-wx`	011	3
`r--`	100	4
`r-x`	101	5
`rw-`	110	6
`rwx`	111	7

Le mode d'un fichier peut alors être spécifié par un nombre en base 8, dont les chiffres représentent, de gauche à droite, les droits d'accès pour :

① le propriétaire du fichier

② les membres du groupe du fichier

③ les autres utilisateurs

745 représente les droits `rwx r-- r-x`

701 représente les droits `rwx --- --x`

333 représente les droits `-wx -wx -wx`

```
{epicea08-beaufils-~/tmp} ls -l
total 4
drwxr-x---    2 beaufils ens          4096 Oct 20 14:50 titi
-rw-r--r-x    1 beaufils ens             0 Oct 20 14:51 toto


{epicea08-beaufils-~/tmp} chmod 745 toto
{epicea08-beaufils-~/tmp} ls -l
total 4
drwxr-x---    2 beaufils ens          4096 Oct 20 14:50 titi
-rwxr--r-x    1 beaufils ens             0 Oct 20 14:51 toto


{epicea08-beaufils-~/tmp} chmod 701 toto
{epicea08-beaufils-~/tmp} ls -l
total 4
drwxr-x---    2 beaufils ens          4096 Oct 20 14:50 titi
-rwx-----x    1 beaufils ens             0 Oct 20 14:51 toto


{epicea08-beaufils-~/tmp} chmod 333 titi
{epicea08-beaufils-~/tmp} ls -l
total 4
d-wx-wx-wx    2 beaufils ens          4096 Oct 20 14:50 titi
-rwx-----x    1 beaufils ens             0 Oct 20 14:51 toto
```

umask

Lorsqu'un programme crée un fichier, il spécifie les droits d'accès qu'il demande pour ce fichier.

Certains des droits demandés seront accordés d'autres seront refusés en fonction d'un *masque de protection*.

La commande `umask` permet :

- ☞ de connaître la valeur du masque si elle est utilisée sans argument
- ☞ de modifier la valeur du masque si elle est utilisée avec un argument

Dans tous les cas elle utilise des masques sous forme numérique octale.

Les droits accordés sont déterminés en retirant aux droits demandés les droits spécifiés par le masque.

Pour les répertoires :

droits demandés :	`rwx rwx rwx`	777
- masque :	`--- -w- rwx`	027
droits accordés :	`rwx r-x ---`	750

Pour les fichiers ordinaires :

droits demandés :	`rw- rw- rw-`	666
- masque :	`--- -w- -w-`	022
droits accordés :	`rw- r-- r--`	644

```
{epicea08-beaufils-~/tmp} umask
0

{epicea08-beaufils-~/tmp} mkdir toto
{epicea08-beaufils-~/tmp} ls -l
total 1
drwxrwxrwx    2 beaufils ens     1024 Nov  6 07:36 toto


{epicea08-beaufils-~/tmp} rmdir toto
{epicea08-beaufils-~/tmp} umask 022
{epicea08-beaufils-~/tmp} mkdir toto
{epicea08-beaufils-~/tmp} ls -l
total 1
drwxr-xr-x    2 beaufils ens     1024 Nov  6 07:37 toto


{epicea08-beaufils-~/tmp} rmdir toto
{epicea08-beaufils-~/tmp} umask 077
{epicea08-beaufils-~/tmp} mkdir toto
{epicea08-beaufils-~/tmp} ls -l
total 1
drwx------    2 beaufils ens     1024 Nov  6 07:37 toto
{epicea08-beaufils-~/tmp} umask
```

Cours n° B.3

Les processus

Définitions

Un **programme** est une suite d'instructions que le système doit faire accomplir au processeur pour résoudre un problème particulier. Ces instructions sont rangées dans un fichier.

Un **processus** correspond au déroulement (*l'exécution*) d'un programme par le système dans un environnement particulier.

$$\begin{array}{ccc} \text{PROGRAMME} & \neq & \text{PROCESSUS} \\ \Updownarrow & & \Updownarrow \\ \text{Recette de cuisine} & \neq & \text{Préparation d'un plat} \end{array}$$

Analogie classique

Soit un informaticien qui prépare un gâteau d'anniversaire pour sa fille.

Il a une recette pour faire le gâteau et dispose de farine, d'œufs, de sucre ...

Ici la recette représente le programme (algorithme traduit en une suite d'instructions), l'informaticien joue le rôle du processeur (CPU) et les ingrédients sont les données à traiter.

Le processus est l'activité de notre cordon bleu qui lit la recette, trouve les ingrédients nécessaires et fait cuire le gâteau.

Si le fils de l'informaticien arrive en pleurant parce qu'il a été piqué par une guêpe, son père marque l'endroit où il était dans la recette (*l'état du processus en cours et son contexte sont sauvegardés*), cherche un livre sur les premiers soins et commence à soigner son fils.

Le processeur passe donc d'un processus (la cuisine) *à un autre plus prioritaire* (les soins médicaux), *chacun d'eux ayant un programme propre* (la recette et le livre des soins).

Lorsque la piqûre de la guêpe aura été soignée, l'informaticien reprendra sa recette à l'endroit où il l'avait abandonnée.

- à l'échelle d'une journée les actions de l'informaticien ont été faites en séquence (les unes après les autres)
- à l'échelle de l'année elles ont toutes été faites en même temps (le même jour !)

➥ pseudo parallèlisme

- Une même activité pourrait avoir été fait dans différents endroits (soins dans la cuisine plutôt que dans la salle de bains par exemple).
- Une même activité aurait pu être faite sur d'autres objets (œufs du voisins par exemple).

➥ notion de contexte d'exécution

UNIX est multitâches : il simule l'exécution simultanée de plusieurs processus grâce à un **ordonnanceur de tâches** (*scheduler*) qui choisit d'exécuter et de basculer d'un processus à un autre très rapidement.

➥ La commutation de tâches permet de simuler le parallélisme d'exécution des processus.

Généralités

- Chaque processus peut lui même démarrer d'autres processus ; dans ce cas le créateur est appelé le père et les processus qu'il a créé sont appelés ses fils.

  ➥ Notion d'arborescence des processus

- Au démarrage du système il n'existe qu'un seul processus qui est donc l'ancêtre de tous les autres (il exécute le programme `init`). Son rôle est de créer 2 type de processus :
  - **interactifs** associés à un terminal particulier
  - **non-interactifs** (*daemons*) rattachés à aucun terminal
- Les processus des utilisateurs sont démarrés par un processus interactif qui exécute un programme particulier : un interpréteur de commandes (*shell*).
- **Le shell démarre un processus pour chacun des ordres (commandes) de l'utilisateur associé.**

Représentation interne

Un processus est une zone mémoire de taille fixe qui permet de stocker :

- ☞ les informations sur le processus lui même
- ☞ le **code** : les instructions à exécuter (dans le langage du processeur)
- ☞ la **zone de données** : les variables manipulées par le code
- ☞ la **pile d'exécution** : les paramètres d'appels des fonctions

Un processus est donc représenté comme un programme qui s'exécute et qui possède son propre compteur ordinal (l'adresse en mémoire de la prochaine instruction à exécuter).

Les informations nécessaires au fonctionnement d'un processus (exécution, arrêt, reprise, etc.) constitue le **contexte d'exécution** de celui-ci.

Contexte d'exécution

Le noyau maintient une table pour gérer l'ensemble des processus. Chaque processus est donc identifié par un index dans cette table :

son numéro d'identification ou **PID**.

Chaque entrée de la table correspond aux informations sur ce processus :

- le numéro d'identification du processus père ..... **PPID**
- l'identifiant de l'utilisateur qui exécute le processus ..... **UID**
- l'identifiant du groupe de l'utilisateur qui exécute le processus **GID**
- le répertoire courant
- la liste des fichiers utilisés par le processus
- le masque de création des fichiers ..... **umask**
- la taille maximale des fichiers que ce processus peut créer .. **ulimit**
- le terminal de contrôle associé
- la zone mémoire associée, ...

Modes de fonctionnement

Si le père n'attend pas la fin du déroulement de son fils pour continuer à travailler on dit que le père crée son fils en **tâche de fond** (*background*) ou mode asynchrone :

➥ les processus s'exécutent en *parallèle*

Si le père ne fait rien tant que son fils n'a pas terminé son programme on dit que le fils est en **avant plan** (*foreground*), ou mode synchrone :

➥ les processus s'exécutent en *séquence*

Pour chaque commande exécutée le shell crée un nouveau processus.

Par défaut le shell exécute les commandes comme des processus en mode synchrone.

- Pour lancer une commande en avant-plan il suffit de taper cette commande :

```
$ commande
... résultat de la commande
$
```

➥ Ce mode est le mode par défaut dans les shells.

- Pour lancer une commande en tâche de fond, il faut faire suivre cette commande par le caractère esperluète «&» :

```
$ commande &
[1] 31343
$
... résultat de la commande
```

Le Bourne shell (`sh`) affiche un numéro de tâche (*job*) entre crochets puis le PID du processus créé avant de rendre la main à l'utilisateur.

Changements d'états

Entrées/Sorties

Tous les processus gère une table stockant le nom des différents fichiers qu'ils utilisent. Chaque index de cette table est appelé un *descripteur de fichiers*.
Par convention les trois premiers descripteurs correspondent à :

0 **l'entrée standard** :
si le programme exécuté par le processus a besoin de demander des informations à l'utilisateur il les lira dans ce fichier (par défaut c'est le terminal en mode lecture).

1 **la sortie standard** :
si le programme a besoin de donner des informations à l'utilisateur il les écrira dans ce fichier (par défaut c'est le terminal en mode écriture).

2 **la sortie d'erreur** :
si le programme a besoin d'envoyer un message d'erreur à l'utilisateur il l'écrira dans ce fichier (par défaut c'est le terminal en mode écriture).

Redirections

En shell, il est possible de modifier les fichiers identifiés par les descripteurs :

- ☞ Redirection de la sortie standard avec le caractère plus grand «>» :
  - ➙ `commande > fichier`
    Si le fichier n'existe pas, il est crée par le shell et s'il existe déjà le shell détruit son contenu pour le remplacer par la sortie de la commande
  - ➙ `commande >> fichier`
    Si le fichier n'existe pas, il est crée par le shell et s'il existe déjà la sortie de la commande est ajoutée à la fin du fichier.
- ☞ Redirection de l'entrée standard avec le caractère plus petit petit «<» :
  - ➙ `commande < fichier`
    La commande lit ses données dans le fichier.

Syntaxe générale des redirections

n<*fichier*	redirige le descripteur numéro *n* en lecture vers *fichier*.
n>*fichier*	redirige le descripteur numéro *n* en écriture vers *fichier*.
n<<*marque*	redirige le descripteur numéro *n* en lecture vers les lignes suivantes jusqu'à ce que la *marque* soit lue.
n>>*fichier*	redirige le descripteur numéro *n* à la fin de *fichier* sans détruire les données préalablement contenues dans ce fichier.
n<&*m*	duplique le descripteur numéro *n* sur le descripteur numéro *m* en lecture, ainsi *n* et *m* seront dirigés vers le même fichier.
n>&*m*	duplique le descripteur numéro *n* sur le descripteur numéro *m* en écriture.

➥ Il est possible de mettre autant de redirections que voulues sur une ligne de commandes.

```
bash-2.08$ ls > resultat

bash-2.08$ cat resultat
fichier
resultat

bash-2.08$ cat <<toto >>resultat
> une ligne en plus
> toto

bash-2.08$ cat resultat
fichier
resultat
une ligne en plus

bash-2.08$ ls toto 1>resultat 2>&1

bash-2.08$ cat resultat
ls: toto: No such file or directory
```

Communication inter-processus

Il est possible d'avoir plusieurs processus fonctionnant en *parallèle* qui communiquent entre eux par le biais de **tubes** (*pipes*). Le système assure alors la synchronisation de l'ensemble des processus ainsi lancés.

Le principe est assez simple :

> La sortie standard d'un processus est redirigée vers l'entrée d'un tube dont la sortie est dirigée vers l'entrée standard d'un autre processus.

Le lancement concurrent de processus communiquant deux par deux par l'intermédiaires des tubes sera réalisé par une commande de la forme :

```
commande1 | commande2 | ... | commandeN
```

➽ Ce mécanisme est une des forces d'UNIX :

un ensemble de petits programmes **fiables** qui communiquent entre eux via le système d'exploitation.

Il existe de nombreuses commandes UNIX qui profitent de ce genre de communication, notamment les *filtres* :

des programmes qui lisent des données sur l'entrée standard, les modifient et envoient le résultat sur la sortie standard

Quelques filtres

`cat`	retourne les lignes lues sans modification.
`cut`	ne retourne que certaines parties de chaque lignes lues.
`grep`	retourne uniquement les lignes lues qui correspondent à un modèle particulier ou qui contiennent un mot précis.
`head`	retourne les premières lignes lues.
`more`	retourne les lignes lues par bloc (dont la taille dépend du nombre de lignes affichables par le terminal) en demandant une confirmation à l'utilisateur entre chaque bloc.
`sort`	trie les lignes lues.
`tail`	retourne les dernières lignes lues.
`tee`	envoie les données lues sur la sortie standard **ET** dans un fichier passé en paramètre.
`tr`	remplace des caractères lus par d'autres.
`uniq`	supprime les lignes identiques.
`wc`	retourne le nombre de caractères, mots et lignes lus.
`sed`	édite le texte lu (requêtes `ed` comme avec la directive «`:`» de `vi`).

➜ Chacune de ces commandes possèdent de nombreuses options décrites dans le manuel.

```
bash-2.08$ getent passwd | grep beaufils | tee /tmp/out | cut -d: -f 5
Bruno.BEAUFILS

bash-2.08$ cat /tmp/out
beaufils:x:1000:1000:Bruno.BEAUFILS:/home/ens/beaufils:/bin/bash

bash-2.08$ ls -l /media/
total 16
drwxr-xr-x  2 root root 4096 2002-12-30 19:28 cdrom
drwxr-xr-x  2 root root 4096 2002-12-30 19:28 floppy
drwxr-xr-x  2 root root 4096 2003-05-16 08:08 usb
drwxr-xr-x  2 root root 4096 2003-11-12 01:43 zip

bash-2.08$ ls -l /media | grep usb | cut -d\  -f1 | cut -c 5-7
r-x
```

```
bash-2.08$ getent passwd \
| grep Laurent | tee /tmp/t1 \
| cut -d: -f 5 | tee /tmp/t2 \
| sort -t. -k2 -r | tee /tmp/t3 \
| tr . ' '

bash-2.08$ cat /tmp/t1
blondel:x:1447:1020:Laurent.BLONDEL:/home/ens_ext/blondel:/bin/bash
behaguel:x:1141:1015:Laurent.BEHAGUE:/home/iutfi2/behaguel:/bin/bash
mulierl:x:1331:1014:Laurent.MULIER:/home/iupqepi3/mulierl:/bin/bash

bash-2.08$ cat /tmp/t2
Laurent.BLONDEL
Laurent.BEHAGUE
Laurent.MULIER

bash-2.08$ cat < /tmp/t3
Laurent.MULIER
Laurent.BLONDEL
Laurent.BEHAGUE
```

Cours n° B.4

Les langages de commandes

Langages de commandes

Un langage de commande (*shell*) est un programme capable d'interpréter des commandes qui seront exécutées par le système d'exploitation.

- ☞ Interface entre le système d'exploitation et l'utilisateur.
- ☞ Permet d'écrire des programmes comportant plusieurs commandes.

Il existe un grand nombre de shells différents séparés, essentiellement par la syntaxe, en 2 grandes familles :

① ceux dérivant du **Bourne-shell** (`/bin/sh`)
Historiquement le premier shell (écrit par Steve Bourne).
Plutôt orienté programmation qu'interaction.

→ **Bourne Again SHell** (`/bin/bash`)
une implémentation du Bourne shell faite par le projet GNU.

→ **Korn SHell** (`/bin/ksh`)
écrit par David Korn.

② ceux dérivant du **C-shell** (`/bin/csh`)
Syntaxe très proche de celle du langage C.
Plutôt orienté interaction que programmation.

→ le **Tenex C-SHell** (`/bin/tcsh`) implémentation libre du C-shell (beaucoup de fonctionnalités dédiées interaction).

➠ nous allons étudier le Bourne Shell via `bash`

Fichier de commandes

Un programme shell (*script*) :
– est une commande constituée d'appel à d'autres commandes shells
– est écrit dans un simple fichier texte :

1. la **première ligne** du fichier définie le *langage* à utiliser :

   `#!<emplacement_du_shell>`

2. le fichier doit être exécutable.
3. le shell doit pouvoir trouver le fichier.
4. le caractère «#» permet d'insérer des commentaires dans le fichier.

Un script est une commande comme une autre sur laquelle on peut faire redirections, tubes, etc.

Paramètres de script (1)

Comme toute commande un script peut être appelé avec des paramètres :
– pour modifier son comportement
– pour spécifier les données qu'il doit manipuler

Chacun des paramètres passés sur la ligne de commandes :
– est repéré par sa position *paramètres positionnels*
– est utilisable dans le script via des *variables*

Paramètres de script (2)

Lors de l'exécution le shell remplace **automatiquement** certains *mots* :

`$0`	le nom du script tel qu'il a été appelé
`$1`	le premier mot apparaissant après le nom du script
`$2`	le second mot apparaissant après le nom du script
⋮	
`$n`	le n^{e} mot apparaissant après le nom du script
`$#`	le nombre de paramètres passés au script
`$*`	**une chaîne** contenant tous les paramètres passés au script (à partir de `$1`) séparés les uns des autres par **un** espace
`"$@"`	autant de chaîne que de paramètres passés au script

`shift`

La commande `shift` permet de décaler les paramètres positionnels vers la gauche.

Par défaut `shift` décale un argument à gauche.

Avec un nombre entier en paramètre `shift` décale plusieurs arguments.

Algorithme classique d'un shell

① Si mode interactif alors envoyer un message (*prompt*) sur la sortie standard

② Lire une ligne de commandes sur l'entrée standard

③ Interpréter cette ligne :

- ❶ **Transformations successives des mots de la ligne** :
  - ☞ Développement des variables
  - ☞ Substitution de commandes
  - ☞ Développement des noms de fichiers
- ❷ **Exécution de la ligne transformée** :
  - ☞ Découpage de la ligne en commandes
  - ☞ Préparation des processus (redirections, etc.)
  - ☞ Pour chacune des commandes :
    - i. Recherche de la commande
    - ii. Exécution ou envoi d'un message sur la sortie d'erreur

④ Retour en ①

Transformations successives de la ligne

La *ligne* lue subies plusieurs développements (*expansion*) avant exécution :

1. Développement des variables
2. Substitution de commandes
3. Découpage des mots
4. Développement des chemins de fichier

Transformations

Les transformations sont effectuées grâce à

- des **caractères spéciaux** (*meta-characters*)
  espace, tabulation,
  «|», «&», «;», «(», «)», «<» et «>»
- des **mots réservés**, notamment :
  - ceux commençant par les caractères : «$», «~», «‘»
  - ceux contenant les caractères : «*», «?», «[», «]», «^», «-»
  - les mots réduits aux caractères : «{», «}»

Tous ces caractères ont donc **un sens particulier** pour le shell

Protections

Il est possible d'empêcher le shell de donner un sens particulier à ces caractères, en les protégeant (*quoting*) :

- **Protection d'un caractère** : faire précéder le caractère à protéger d'une barre de fraction inversée (*backslash*) «\»
  ➠ le caractère protégé est laissé tel quel sur la ligne finale, le backslash est supprimé.

- **Protection complète** : entourer la zone à protéger par des guillemets simples (*quote*) «'»
  ➠ aucune transformation n'est faite à l'intérieur de la zone protégée.
- **Protection simple** : entourer la zone à protéger par des guillemets doubles (*double-quote*) «"»
  ➠ ormis «\», «$» et «`» aucun caractère spécial n'est plus interprété.

Variables

En shell, comme dans tous langages de programmation il existe une notion de variables avec quelques spécificités :

- ☞ Les noms de variables sont des identificateurs ne comprenant que des lettres, des chiffres ou le caractère de soulignement «_».

- ☞ Il n'existe pas de types de variables
  ➠ toutes les valeurs sont considérées comme des suites de caractères

- ☞ Il n'y a pas de réévaluation des variables
  ➠ une variable ne peut être modifiée que par une affectation

Affectation des variables

- **Variables classiques** :
  elles ne sont définies que dans le contexte d'exécution du processus dans lequel elles sont déclarées.

  `<nom>=<valeur>`

- **Variables d'environnement** :
  elles sont définies dans le contexte d'exécution du processus dans lequel elles sont déclarées et dans tous les contextes d'exécution des processus que celui-ci peut créer (processus fils)

  `<nom>=<valeur>`
  `export <nom>`

Développement des variables

`${<nom>}`

- ☞ le shell substitue la variable par la dernière valeur qui lui a été affecté
- ☞ les accolades «{» et «}» sont optionnelles, elles sont cependant souvent utilisées pour délimiter le nom de la variable
- ☞ si la variable n'existe pas le shell substitue par une chaîne vide

Quelques variables particulières

variables	description
`$HOME`	Le chemin absolu du répertoire principal de l'utilisateur.
`$PATH`	Liste des répertoires dans lesquels le shell recherche les commandes à exécuter. Les répertoires sont séparés par des deux-points « : ».
`$?`	Le code de retour de la dernière commande exécutée.
`$$`	Le PID du processus exécutant le shell en cours.
`$PPID`	Le PID du processus père du processus `$$`.
`$!`	Le PID du dernier processus exécuté en tâche de fond.
`$PWD`	Le répertoire de travail en cours.
`$PS1`	Le message d'invite (*prompt*) principal du shell.
`$PS2`	L'invite secondaire du shell.

Substitution des commandes

Il est possible de remplacer un bout de la ligne de commande par le résultat de l'exécution d'une commande :

① la zone représentant la commande à exécuter doit être entourée de guillemet inverse (*back-quote*) «‘»

② un processus fils (sous-shell) exécutant la commande située dans la zone entourée est créé

③ la sortie standard de ce processus est capturée et remplace la zone sur la ligne de commande.

```
echo 2 + 3 = ‘expr 2 + 3‘
```

⇓

```
echo 2 + 3 = 5
```

⇓

```
2 + 3 = 5
```

Développement des chemins de fichiers

Un **joker** est un caractère utilisé *à la place* d'un (de plusieurs) autre(s).

Un **modèle** (*glob pattern*) est un mot qui contient un ou plusieurs jokers.

Lorsque le shell trouve un modèle sur la ligne de commande :

1. il cherche la liste des fichiers dont le **chemin correspond** au modèle
2. il remplace le modèle par les **chemins des fichiers** correspondant en les séparant par **un** espace.
3. si aucun fichier ne correspond le modèle est laissé tel quel

Jokers (Méta-caractères)

- «*» remplace n'importe quelle **suite de caractères** (y compris vide)
- «?» remplace n'importe quel **caractère**
- «[» <*liste*> «]» remplace n'importe quel caractère de <*liste*>
  - on spécifie la liste des caractères que l'on veut représenter
  - «^» placé en début de liste signifie que l'on veut remplacer n'importe quel caractère **non présent** dans la liste
  - «-» utilisé dans la liste définit un intervalle plutôt qu'un ensemble de valeurs

Exemples de modèles de chemins

`f*`	Tous les fichiers dont le nom commence par `f`
`f?`	Tous les fichiers dont le nom fait 2 caractères et commence par `f`
`*.java`	Tous les fichiers dont le nom se termine par `.java`
`tit[oi]`	Les fichiers `titi` et `tito`
`[a-z]*[0-9]`	Tous les fichiers dont le nom commence par une minuscule et se termine par un chiffre
`[^a-z]*`	Tous les fichiers dont le nom **ne commence pas** par une minuscule
`???*`	Tous les fichiers dont le nom est composé d'au moins 3 caractères.

Découpage de la ligne en commandes

- Les mots sont séparés par des **blancs** non protégés

  Un blanc est un caractère espace ou une tabulation

- Une commande est un mot quelconque :
  - situé en **première position** de la ligne
  - ou situé **juste après un séparateur** de commandes
  - éventuellement **suivies** par des paramètres (d'autres mots)

- Les commandes peuvent être séparées par :
  - des **points-virgules** «;»

    Attendre la fin d'une commande avant de passer à la suivante
  - des **esperluètes** «&»

    Ne pas attendre la fin d'une commande pour passer à la suivante
  - des **tubes** «|»

    Démarrer les commandes en parallèle en les connectant

Opérateurs

Autres séparateurs possibles : opérateurs logiques séquentiels paresseux

- **ET** `cmd1 && cmd2`

  `cmd2` est exécutée si et seulement si `cmd1` a réussi

- **OU** `cmd1 || cmd2`

  `cmd2` est exécutée si et seulement si `cmd1` a échoué

- **Le shell essaie de faire réussir la ligne**
  - évaluation de gauche à droite
  - dès qu'on sait que la séquence ne peut pas réussir (ou qu'elle est déjà réussie) on arrête son évaluation

Définition de réussite (échec) dans le manuel

Recherche et exécution de la commande

1. Si la commande est interne elle est exécutée directement

2. Sinon
   (a) Recherche répertoire et fichier
   - si la commande contient au moins un caractère «/»
     extraction du répertoire et du nom de fichier
   - sinon pour tous les répertoires définis dans la variable `$PATH`
     recherche d'un fichier correspondant à la commande

   (b) Exécution ou erreur
   - si un fichier a été trouvé **et qu'il est exécutable**
     exécution du code qu'il contient dans un nouveau processus
   - sinon envoi d'un message d'erreur

Commandes internes

Les commandes internes (*builtins commands*) sont traités directement :

- ☞ pas de nouveaux processus pour les exécuter
- ☞ leur code est intégré au shell
- ☞ peuvent modifier le contexte d'exécution du shell courant

commandes	**description**
`cd`	change le répertoire courant
`echo`	envoie ses arguments sur la sortie standard
`pwd`	envoie le nom du répertoire courant sur la sortie standard
`.` ou `source`	lit et exécute les commandes d'un fichier
`exec`	remplace le code par une autre commande
`exit`	termine le processus courant
`read`	affecte une variable en lisant l'entrée standard

Documentées dans la page du manuel de `bash`

Commandes externes

- ☞ code stocké dans un **fichier régulier exécutable**
- ☞ rangée dans un répertoire de la hiérarchie du système
  la convention est d'utiliser des répertoire nommés `bin`
- ☞ recherchée dans une liste de répertoires
  répertoires séparés par des « : » dans la variable `PATH`
- ☞ exécutée dans un nouveau processus par le shell
- ☞ elles **ne peuvent pas** modifier le contexte d'exécution du shell

Quelques exemples :

`/bin/ls`, `/bin/cp`, `/bin/mv`, `/bin/mkdir`, `/usr/bin/vi`,
`/usr/local/bin/as400j`

Structure pour

```
for var in liste
do
    cmds
done
```

- `cmds` exécutés autant de fois qu'il y a d'élément dans `liste`
- Pour chaque tour `$var` a comme valeur un des éléments de `liste`.
- Éléments utilisés de la gauche vers la droite de la liste
- `liste` est définie après les développements du shell

Code de retour

Sous UNIX toutes les commandes ont un code de retour :

- ☞ invisible sur la sortie standard
- ☞ visible pour le shell via une variable (`$?`)
- ☞ convention :
  - ☞ si la commande **réussit** le code de retour **vaut 0**
  - ☞ si elle **échoue** le code de retour **est différent de 0**

➠ possibilité de faire des actions conditionnées au résultat d'autres actions

$$\text{Vrai} \equiv \textbf{réussite} \equiv \text{code de retour} = 0$$
$$\text{Faux} \equiv \textbf{échec} \equiv \text{code de retour} \neq 0$$

Structure si

```
if cmd-si
then
    cmdsif
elif cmd-sinon-si
    cmdselif
else
    cmdselse
fi
```

1. si `cmd-si` réussit alors `cmdsif` exécutés
2. sinon
   (a) si `cdm-sinon-si` réussit alors `cmdselif` exécutés
   (b) sinon `cmdselse` est exécutés

Structure tant que

```
while cmd-tq
do
     cmdswhile
done
```

① `cmd-tq` est exécutée

② si elle a réussit

  ❶ les `cmdswhile` sont exécutées

  ❷ retour en ①

Modes de fonctionnement

Tous les shells ont 3 modes de fonctionnement :

1. login interactif ..... (connexion à la machine)
2. shell interactif ..... (appel de `bash`)
3. shell non-interactif ..... (script)

Initialisation

Certaines commandes sont exécutées au démarrage de chacun des modes :

1. dans le processus d'un login interactif
   (a) les commandes du fichier `/etc/profile` sont lues et exécutées
   (b) les commandes du fichier `${HOME}/.bash_profile` sont lues et exécutées
   (c) les commandes du fichier `${HOME}/.profile` sont lues et exécutées
2. dans le processus d'un shell interactif les commandes du fichier `${HOME}/.bashrc` sont lues et exécutées
3. dans le processus d'un shell non interactif si la variable `BASH_ENV` a une valeur le shell considère que son contenu (après transformation) est le nom d'un fichier dont les commandes doivent être lues et exécutées