Support du cours

# Système d'exploitation UNIX/Linux

# -

# Seconde partie

# -

# Scripting en bash (et en (t)csh)

# -

# Scripting en Python (et en Perl)

Pierre Pompidor

# Table des matières

# 1 Introduction au scripting système :

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

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

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

- le langage de programmation associé à **bash**
- le langage de programmation associé à **(t)csh**
- le langage **Python**
- le langage **Perl**
- le langage **Ruby**

Les langages bash et (t)csh ont le gros avantage d'interfacer directement les commandes et les gros inconvenients de ne pas être modulaires et d'avoir des syntaxes vieillottes et piégeuses (surtout bash) : de ces deux langages, seul bash sera abordé en cours.
Des langages de plus haut niveau et universels (car ils peuvent être utilisés pour d'autres buts que la programmation système) Python, Perl et Ruby, seul Python sera abordé en cours (et sans utiliser le paradigme de la programmation par objets).

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

# 2 Scripts exécutés par l'interpréteur de commandes ((t)csh ou bash) :

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

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

- **interactivement** : les instructions du programme sont directement saisies dans le terminal
  → un nouveau prompt apparaît si le bloc d'instructions n'est pas fini (par exemple les instructions comprises dans une boucle).
- sous la forme d'**un programme exécutable** (généralement suffixé par **.sh**) :
  par exemple, le script nommé *bonjour.sh* contenant la ligne

  ```
  echo bonjour
  ```

  sera exécuté par l'une des commandes suivantes :
  - `./bonjour.sh` si le script est exécutable (le point fait référence au répertoire courant) ;
  - `bonjour.sh` si le script est exécutable et si la variable d'environnement **PATH** contient le nom du répertoire où le script se trouve ;
  - `source bonjour.sh` ou `. bonjour.sh` si le script n'est pas forcément exécutable ;
  - `bash bonjour.sh` pour (bizarrement) exécuter ce script par un autre processus bash que celui qui gère le terminal.

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

Si vous voulez explicitement préciser dans un script shell, quel interpréteur de commandes doit l'exécuter, vous pouvez faire apparaître un shebang ;) sur la première ligne du script :
- `#!/bin/csh` pour **csh** (ou tcsh)
- `#!/bin/bash` pour **bash**

## 2.1 La configuration de bash ou de (t)csh :

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

Voici les principaux fichiers de configuration exécutés :
- Sous **csh** ou **tcsh** :
  - */etc/csh.cshrc*
  - */etc/csh.login*
  - **.cshrc** (dans votre répertoire d'accueil) → ce fichier va contenir vos configurations
- Sous **bash** :
  - */etc/profile*
  - *.bash_profile* (dans votre répertoire d'accueil)
  - **.bashrc** (dans votre répertoire d'accueil) → ce fichier va contenir vos configurations

Ces fichiers de configuration vous permettent de :
- compléter la variable d'environnement **PATH** qui liste tous les répertoires dans lesquels des exécutables peuvent être retrouvés par l'interpréteur de commandes :
  par exemple, voici comment la compléter avec le répertoire courant

  **Sous csh/tcsh** : *setenv PATH "${PATH} :."*
  **Sous bash** : *export PATH=$PATH :.*

- de définir des alias qui sont des synomymes à une commande :

  **Sous csh/tcsh** : alias nouvelle_commande 'ancienne_commande'
  *alias c 'clear'*
  *alias bye exit*

  **Sous bash** : alias nouvelle_commande='ancienne_commande'
  *alias c='clear'*
  Pour créer des alias avec paramètres, il faut créer des fonctions :
  exemple : `alias chm='function \_chm () \{ chmod \$1 \$2; \}; \_chm'`

- changer la valeur du prompt :
  - Sous csh/tcsh, par exemple avec **set prompt='%~ $ '**
    - %~ : nom du répertoire courant
    - %t : heure
  - Sous bash, il faut insérer des codes spéciaux dans la valeur de la variable **PS1** :
    - \d : date
    - \h : nom de la machine
    - \W : nom du répertoire courant
    - \w : chemin du répertoire courant
    - \u : login de l'utilisateur

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

## 2.2 Variables et paramètres :

**Les variables ne sont pas typées.**
L'accès à la valeur d'une variable est signifié en faisant précéder le nom de la variable par $.

### 2.2.1 Les variables système :

Des variables prédéfinies permmettent d'accéder (et de modifier) des informations relatives au système d'exploitation :
- les **variables système d'environnement** dont les valeurs seront accessibles par les programmes lancés sous bash ou tcsh ;
- les variables système (simples) dont les valeurs n'ont une importance que pour l'interpréteur de commandes.

Voici les principales variables d'environnement système :

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

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

### 2.2.2 Déclaration / initialisation des variables :

Avec bash :

```
nom_variable=valeur
```

Exemple :

```
prenom=Pierre
```

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

Une fois définie, une variable doit être préfixée par **$** pour être utilisée !

```
echo $prenom
```

Une variable peut être instanciée par le résultat d'une commande :

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

Exemple :

```
ll=‘ls -l‘
```

Dans le cas de l'instanciation d'une variable d'environnement, il faut utiliser la commande **export**.
Exemple (ici pour rajouter le répertoire courant dans le PATH) :

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

Enfin pour récupérer une valeur au clavier, il faut utiliser la commande **read** :

```
read entree
echo $entree
```

Avec (t)csh :

Déclaration / initialisation d'une variable :
`set nom_variable = valeur`
Une variable peut être instanciée par le résultat d'une commande : `set variable = ‘commande‘`
Dans le cas de l'instanciation d'une variable d'environnement, il faut utiliser la commande **setenv**.

### 2.2.3 Calcul arithmétique :

Sous *(t)csh* avec @ : `@ nom_variable = expression arithmétique`
Sous *bash* :
- avec la syntaxe `$((<opération>))` : `i=$((3+4)); echo $i`
- avec la commande interne `let` : `let i=3+4; echo $i`

### 2.2.4 Les paramètres de la ligne de commande :

Les paramètres de la ligne de commande sont accessibles par les variables suivantes :

| | (t)csh | bash |
|---|---|---|
| nombre de paramètres | `$#argv` | `$#` |
| liste des paramètres | `$argv` | `$*` |
| nom du script | `$argv[0]` | `$0` |
| nième paramètre (raccourci $n) | `$argv[n]` | `$n` |

## 2.3 Structures de contrôles des scripts :

### 2.3.1 Structure conditionnelle avec if :

Sous *bash* et comme dans la quasi totalité des langages de programmation, le mot réservé **if** introduit une expression conditionnelle dans une structure de programmation qui permet de conditionner l'exécution d'un bloc d'instructions. Mais à la différence des autres langages de programmation, l'expression conditionnelle peut être encadrée par différents délimitateurs (crochets, doubles crochets, parenthèses) ou même ne pas en utiliser : cette diversité est affreuse. Vraiment.

Voici en apéritif, quelques exemples d'expressions conditionnelles :

```
if ls -l | grep -qc Cours -> l'expression conditionnelle est une ligne de commande :
                             pas de délimitateurs (l'option q de grep l'empêche d'afficher son résultat)
if test $i -gt 0          -> comparaison arithmétique avec la commande test et les opérateurs classiques
                             ici -gt teste que le premier opérande est plus grand que le second
if [ $i -gt 0 ]           -> les crochets remplacent la commande test
                             il faut mettre une espace après le premier crochet et avant le dernier
if [[ $i > 0 ]]           -> Les doubles crochets permettent d'utiliser tous les opérateurs arithmétiques
                             mais attention, la comparaison est lexicographique (voir exemple suivant)
if [[ $a > $b ]]          -> si $a vaut 20 et $b vaut 100, ce test renvoie vrai car 2 est supérieur à 1 !
if (( $a > $b ))          -> avec les doubles parenthèses, la comparaison arithmétique fonctionne !
```

Revenons au schéma d'utilisation de la structure conditionelle avec if (et ici en encadrant l'expression conditionnelle avec des doubles crochets) :

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

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

```
if [[ expression conditionnelle ]]; then bloc d'instructions; fi
```

**Les doubles crochets autour de l'expression conditionnelle permettent d'utiliser les opérateurs de comparaisons avancés.**

Il est donc aussi possible :

- d'utiliser la commande *test* ou des simples crochets pour :
  - des tests via des opérateurs unaires, exemple : `if [ -f $fichier ]`
  - des tests via des opérateurs binaires mais "vintage" (*-eq*, *-ne*, *-gt*, *-ge*, *-lt*, *-le*)
    exemple : `if [ $i -eq 1 ]`
- d'utiliser des doubles parenthèses pour des comparaisons arithmétiques
- de ne pas encadrer l'expression conditionnelle dans le cas de l'évaluation du résultat d'une commande directement évaluée dans celle-ci :
  `if <commande>`

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

Sous *(t)csh* les expressions conditionnelles sont sagement circonscrites par des parenthèses :

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

### 2.3.2 case et switch :

Sous *bash* :

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

Sous *(t)csh* :

```
switch (valeur d'une variable)
        case ...:
                bloc d'instructions
        breaksw
        ...
        default :  (facultatif)
                bloc d'instructions
endsw
```

### 2.3.3 Tant que (while) :

Sous *bash* (avec l'expression conditionnelle circonscrite avec des doubles crochets) :

```
while [[ expression conditionnelle ]]
do
        bloc d'instructions
done
```

Sous *(t)csh* :

```
while (expression conditionnelle)
        bloc d'instructions
end
```

### 2.3.4 Itération sur les valeurs d'une liste (foreach, for)

Sous *bash* :

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

La liste peut être :

- une suite de valeur → `for i in 1 2 3; do echo $i; done`
- **produite par l'exécution d'une commande** → `for i in $(cat fichier); do echo $i; done`

Sous *(t)csh* :

```
foreach variable (liste de valeurs)
        bloc d'instructions
end
```

## 2.4 Opérateurs :

Les opérateurs du C peuvent être utilisés à la fois pour bash et (t)csh (mais attention bash possède aussi d'autres "vieux" opérateurs) :

- ==, !=, <, <=, >, >=
- !
- +, -, *, /
- ++, –

Un type d'expression spécifique permet de tester le statut d'un fichier.
Ce type d'expression a la forme générale : **-spécification référence**

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

Bash permet aussi d'utiliser des opérateurs "old-school" du type (ici pour un test d'égalité) `if [ $a -eq $b ]`.

## Divers :

| | |
|---|---|
| **echo** : | affichage à l'écran de ce qui suit (**-n** pour ne pas passer à la ligne) |
| **exit entier** : | sortie du script avec instanciation de la variable système **status** |
| **$ ? pour bash et $status** pour (t)csh : | valeur retournée par le dernier processus |

## 2.5 Exemples de scripts bash :

### 2.5.1 Lister tous les répertoires du répertoire courant :

```
#!/bin/bash
for i in *
do
     if [ -d $i ]
     then
          echo $i est un répertoire
     fi
done
```

### 2.5.2 Afficher le nom de tous les fichiers réguliers qui contiennent une des chaînes de caractères passées en paramètres :

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

```
for f in *
do
  for p in $*
  do
    if [ -f "$f" ]
    then
       r=‘grep -c $p "$f" 2> /dev/null‘
       if [[ $r > 0 ]]
       then
          echo "$f contient $i"
       fi
    fi
  done
done
```

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

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

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

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

## 2.6 Exemples de scripts (t)csh :

### 2.6.1 Lister tous les répertoires du répertoire courant :

```
#!/bin/csh
set liste=‘ls‘
foreach i ($liste)
     if (-d $i) then
          echo $i est un répertoire
     endif
end
```

### 2.6.2 Lister tous les paramètres passés au script :

```
#!/bin/csh
echo Le script s\’appelle $0
echo Il a $#argv parametres qui sont : $argv
echo "A l’envers :"
set i=$#argv
while ($i > 0)
      echo $argv[$i]
      @ i--
end
```

### 2.6.3 Faire la somme des paramètres passés au script :

```
#! /bin/csh
clear
set resultat = 0
foreach i ($argv)
        @ resultat = $resultat + $i
end
echo Le résultat est $resultat
```

### 2.6.4 Comptage des fichiers réguliers dans la sous-arborescence passée en paramètres) :

```
#! /bin/csh
cd $1
set cumul = 0
set liste = ‘ls‘
foreach i ($liste)
        if (-d $i) then
                compte $i  # Appel récursif du script compte
                @ cumul = $cumul + $status
        else
                @ cumul++
        endif
end
echo il y a $cumul fichiers dans ‘pwd‘
exit $cumul
```

# 3 Ecriture de scripts PYTHON et PERL :

## 3.1 Introduction :

**Python** et **Perl** sont les deux langages interprétés les plus adéquats pour réaliser des scripts système, portables sur Unix et sur Windows. Tous les deux gèrent automatiquement la mémoire et sont très conciliants sur les déclarations de types (Perl d'ailleurs beaucoup plus que Python).
De plus en plus Python est utilisé pour l'écriture de scripts système à la place de Perl qui lui même avait (plus ou moins) frappé d'obsolescence les scripts (t)csh ou bash. En effet, synthèse surprenante de C, d'interpréteurs de commandes SHELL et d'utilitaires UNIX (sed, awk ...), Perl est un langage d'une syntaxe difficile. Ainsi, bien que moins "confortable" sur l'application d'expressions régulières, Python lui est de plus en plus préféré pour sa syntaxe objets "propre" (mais néanmoins imparfaite pour les puristes "objets").

**Attention, seuls quelques points des langages Python et Perl sont abordés ici, correspondant à la problématique des scripts système :**
- **opérations de lecture et d'écriture dans des fichiers ;**
- **manipulation de tableaux (listes et dictionnaires) ;**
- **accès au système de fichier et exécution de commandes/programmes externes ;**
- **applications d'expressions régulières.**

**La définition de classes ne sera pas, par exemple, décrite dans ce polycopié !**

## 3.2 Comment exécuter un script

Il y a deux modes de lancement possibles pour un script Python ou Perl :
- en le passant en paramètre à l'interpréteur Python ou à l'interpréteur Perl :

  **Python :** python3 script_Python.py

  **Perl :** perl script_Perl.pl
- en l'exécutant directement sur la ligne de commande à condition que le script soit exécutable et qu'il comporte une première ligne particulière qui permette à l'interpréteur de commandes de savoir à qui il doit déléguer le script :

  **Python : #!/usr/bin/python3** ou mieux encore **#!/usr/bin/env python3**

  **Perl : #!/usr/bin/perl** ou mieux encore **#!/usr/bin/env perl**

Que ce soit en Python ou en Perl, il est également possible d'exécuter des instructions de manière interactive sous l'interpréteur (tapez respectivement **Python** ou **Perl** dans le terminal).

En entête d'un script Python (mais après la première ligne indiquée ci-avant), il est recommandé d'insérer la ligne suivante pour pouvoir utiliser des caractères accentués (ici en UTF8) : `# -*- coding: utf-8 -*-`

Le script peut être accompagné de paramètres ou inséré dans une ligne de commandes :

| | |
|---|---|
| directement : | **script** |
| directement avec paramètres : | **script paramètre_1 paramètre_2 ...** |
| avec en entrée le contenu d'un fichier : | **cat fichier \| script** |
| avec en entrée le contenu d'un fichier et des paramètres : | **cat fichier \| script paramètre_1 paramètre_2 ...** |

Les paramètres d'un script sont accessibles :
En Python dans la liste **sys.argv** à condition d'avoir importer le module *sys* (voir ci-après)
En Perl dans le tableau **@ARGV** (voir ci-après)

Que ce soit en Python ou en Perl, les lignes (ou fin de lignes) en commentaires sont précédées par #

## 3.3 Définitions des blocs d'instructions :

En Python :
Les instructions qui définissent les blocs (gouvernés par des structures conditionnelles ou itératives) doivent débuter sur la même colonne. Par ailleurs les expressions conditionnelles n'ont pas besoin d'être encadrées par des parenthèses.

Voici l'exemple d'un test ("if") inséré dans une boucle ("for") :

```
for ligne in lignes:
    res = re.search(r"^(.*)\s+verbe", ligne)  # application d'une expression régulière
    if res:
        print(res.group(1))
```

En Perl :
Les instructions qui définissent les blocs doivent être encadrées par des accolades (très classiquement).
Voici la traduction de l'exemple précédent en Perl :

```
for (@lignes) {
    if (/^(.*)\s+verbe/o) {  # application d'une expression régulière
        print "$1\n";
    }
}
```

Le test "if" et la boucle "for" seront décrits plus loin.

## 3.4 Les variables :

### 3.4.1 Les variables individuelles (ou scalaires) :

En Python :
- Elles ne sont pas typées
  - chaine = "chaîne"
  - reel = 3.14
- Les chaînes de caractères peuvent être encadrées soit par des guillemets, soit par des quotes simples, ce qui permet, par exemple, l'inclusion de quotes entre guillemets :
  chaine = "J'aime les brocolis"
- Le transtypage n'est pas automatiquement effectué suivant le contexte :
  variable = '123' ; print(int(variable) + 1)

En Perl :
- Leurs noms sont **toujours** préfixés par **$**
- Elles ne sont pas typées
  - $chaine = "chaîne" ;
  - $reel = 3.14 ;
  - $resultat_de_commande = `pwd` ;
- Les chaînes de caractères peuvent être encadrées soit par des guillemets, soit par des quotes simples, mais dans le cas où elles sont encadrées par des guillemets, elles autorisent l'interpolation de variables :
  $var1 = 'vous' ;
  $var2 = "Bonjour $var1" ;
- Le transtypage est automatiquement effectué suivant le contexte :
  $variable = '123' ; print $variable + 1, "\n" ;

### 3.4.2 Les listes (appelées tableaux simples en Perl) :

Les tableaux simples sont des tableaux d'éléments indicés par des entiers à partir de 0.

En Python :
- Elles doivent être initialisées (au pire à rien : liste = [])
- Elles peuvent être initialisées par une liste de valeurs :
  jours = ['dimanche', 'lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi']
- Longueur d'une liste (appelée ici L) : len(L)
- Tranche d'une liste : L[i :j] (attention, j est l'indice qui suit le dernier élément de la tranche)
- Concaténation de listes : L1 + L2
- Tri d'une liste : L.sort()
- Inversion d'une liste : L.reverse()

En Perl :
- Elles sont préfixées par **@**
- Elles peuvent être initialisées par une liste de valeurs :
  `@jours = ('dimanche', 'lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi')`
- ($sunday, $monday, ...) = @jours ;
- `$#nom_tableau` donne l'indice du dernier élément du tableau
- **@ARGV** est instancié par les paramètres du script (**$ARGV[0]**, ...)
- Elles peuvent être créées à la volée

Quelques fonctions prédéfinies sur les listes en Perl :

| | |
|---|---|
| extraction du premier élément | **shift** tableau |
| insertion en tête | **unshift** tableau, éléments |
| extraction du dernier élément | **pop** tableau |
| insertion en queue | **push** tableau, éléments |
| tri d'un tableau | **sort** tableau |

### 3.4.3 Les dictionnaires (appelés tableaux associatifs en Perl) :

Les éléments des dictionnaires sont **indexés** par des chaînes de caractères au lieu d'être indicés.

En Python :
- Ils doivent être initialisés (au pire à rien : dictionnaire = {})
- Ils peuvent être initialisés par une liste de couples de valeurs (clef, valeur) :
  mois = { 'janvier' : 31, "février" : 28, ... }
- Pour accéder à la valeur d'un élément, donnez sa clef entre crochets :
  print(mois['janvier'])
- Liste des clefs :
  - directement, exemple : *for clef in mois :*
  - avec la méthode **keys()** : noms_mois = mois.keys()
- Liste des valeurs avec la méthode **values()** : nb_jours = mois.values()
- Test de l'existence d'une clef : **has_key(clef)**

En Perl :
- Ils sont préfixés par **%**
- Initialisation par liste de couples de valeurs (clef, valeur) :
  %mois = ('janvier' => 31, 'février' => 28, ... )
- Exemple d'utilisation : $mois{janvier} ou $mois{'janvier'}

Quelques fonctions prédéfinies sur les listes en Perl :

| | |
|---|---|
| accès aux clefs | avec la fonction **keys** tableau_associatif<br>`foreach $clef (%tableau_asso) {print "$clef: $tableau_asso{$clef}\n";}` |
| accès aux valeurs | **values** tableau_associatif |
| accès au binôme clef/valeur | ($clef, $valeur) = **each** tableau_associatif<br>`while (($clef, $valeur) = each %tableau_asso) {print "$clef: $valeur\n";}` |

## 3.5 Les structures conditionnelles et itératives :

### 3.5.1 Les structures conditionnelles :

En Python :

**if** *expression conditionnelle* **:**
<tabulation> *instruction*
<tabulation> *instruction*
...
**else :**
<tabulation> *instruction*
<tabulation> *instruction*
...

La réalisation d'un SWITCH peut être effectué par l'instruction **elif** :
if *expression conditionnelle* **:**
<tabulation> *instruction*
<tabulation> *instruction*
<tabulation> *...*
**elif** *expression conditionnelle* :
<tabulation> *instruction*
<tabulation> *instruction*
<tabulation> *...*

En Perl :

**if** ( *expression conditionnelle* )
{ ... }
else
{ ... }

La réalisation d'un switch peut être effectué par l'utilisation de labels :

```
SWITCH :
{
        if ( ... ) { ... ; last SWITCH ; }
        if ( ... ) { ... ; last SWITCH ; }
        ...
}
```

*Pas d'équivalence du switch C/JAVA*
*Les labels (ici SWITCH) et la commande* ***last*** *sont présentés plus loin*

### 3.5.2 Les structures itératives :

En Python :

Les boucles sont réalisées soit par un "while", soit par un "for" :
**while** *expression conditionnelle* **:**
<tabulation> *instruction*
<tabulation> *instruction*
<tabulation> *...*

**for** *variable* in *liste de valeurs* **:**
<tabulation> *instruction*
<tabulation> *instruction*
<tabulation> *...*

Exemple sur la liste des jours avec un “while” :

```
jours = ['dimanche', 'lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi']
num_jour = 0
while num_jour < 7 :
    print(jours[num_jour])
    num_jour = num_jour + 1
```

Exemple sur la liste des jours avec un “for” :

```
jours = ['dimanche', 'lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi']
for jour in jours :
    print(jour)
```

En Perl :

**while** (*expression conditionnelle*)
{ ... }

*Un cas fréquent : la recherche d'un motif sur les lignes d'un fichier passé sur l'entrée standard :*
*while (<STDIN>)*
{
*if (/expression régulière/)* {...}
}

**for** (initialisation de l'indice ; expression conditionnelle ; incrémentation/décrémentation)
{ ... }

**for** *($i=0 ; $i < 10 ; $i++)*
{ ... }

**foreach** $variable (*liste de valeurs*)
{ ... }

*Traitement sur une liste d'utilisateurs :*
*foreach $user (@users)*
{...}

### 3.5.3 Les commandes de contrôle des structures itératives :

En Python :

| | |
|---|---|
| **break** | sort de la boucle |
| **continue** | retourne au début de la boucle |

En Perl :

| | |
|---|---|
| **chaîne de caractères :** | pose une marque (un label) |
| **goto** | permet de sauter au label concerné |
| **last** | sortie immédiate de la boucle |
| **next** | saut immédiat à la prochaine itération |
| **redo** | saut immédiat à la prochaine itération sans évaluer l'expression conditionnelle |

## 3.6 Les opérateurs de comparaison :

En Python :

| nombres ou chaînes | |
|---|---|
| == | vrai si le premier opérande est égal au second |
| != | vrai si le premier opérande n'est pas égal au second |
| < | vrai si le premier opérande est inférieur au second |
| > | vrai si le premier opérande est supérieur au second |
| <= | vrai si le premier opérande est inférieur ou égal au second |
| >= | vrai si le premier opérande est supérieur ou égal au second |

En Perl :

| nombres | chaînes | |
|---|---|---|
| == | eq | vrai si le premier opérande est égal au second |
| != | ne | vrai si le premier opérande n'est pas égal au second |
| < | lt | vrai si le premier opérande est inférieur au second |
| > | gt | vrai si le premier opérande est supérieur au second |
| <= | le | vrai si le premier opérande est inférieur ou égal au second |
| >= | ge | vrai si le premier opérande est supérieur ou égal au second |
| <=> | cmp | 0 si égal, 1 si le premier opérande plus grand, -1 si c'est le second |

Un type d'expression spécifique permet de tester le statut d'un fichier.
Ce type d'expression a la forme générale : **-spécification référence** :
**d** type répertoire, **f** type ordinaire (fichier régulier), **T** type fichier texte, **e** existence du fichier,
**r** droit de lecture, **w** droit d'écriture, **x** droit d'exécution

## 3.7 Les expressions régulières :

Une expression régulière définit une chaîne de caractère quelconque qui va être recherchée dans une autre chaîne de caractères (qui peut être la valeur d'une variable, d'une ligne d'un fichier ...).

En Python :
- Le module **re** doit être importé : **import re**
- Pour plus d'efficacité les expressions régulières peuvent être compilées avec la méthode **compile**
- Elles sont appliquées avec la méthode **search()** :
  *resultat = re.search(expression régulière, cible)*
- Les motifs retrouvés sont accessibles avec la méthode **group()** appliquée sur le nom de la variable résultat :
  - group(1) → première sous-chaîne capturée correspondant au premier sous-motif de l'expression régulière indiquée par le premier couple de parenthèses ;
  - group(2) → seconde sous-chaîne capturée correspondant au second sous-motif de l'expression régulière indiquée par le second couple de parenthèses ;
  - ...
- Plusieurs sous-motifs peuvent être capturés par la méthode **findall()**, exemple :
  *nombresEntiers = re.findall("(\d+)", cible)*
  si cible vaux "1 45hj67 6", *nombresEntiers* va valoir ['1', '45', '67', '6']

Dans cet exemple, la chaîne de caractères suivant le mot "endives" va être capturée :

```
recherche = re.compile(r"endives (.*)")  # l'expression régulière est compilée
res = re.search(recherche, "J'aime aussi les endives au jambon")
if res:
    print(res.group(1))  # le motif extrait est "au jambon"
```

En Perl :

Par défaut, elle s'applique sur la variable $_ (qui contient notamment la dernière ligne lue sur un fichier).
Les expressions régulières sont encadrées par des /.
La variable $& va contenir la chaîne de caractères appariée.
Les expressions régulières sont appliquées à une variable par le "binding opérator" :

| | |
|---|---|
| expression appliquée sur une variable | `$variable =~ /expression régulière/` |
| comptage du nombre d'appariement | `$variable = /expression régulière/` |
| motifs appariés dans un tableau | `($var1, $var2, ...) = /...(...)...(...)...../` |
| | `@tab = /...(...)...(...)...../` |
| exemple | `($membres_des_groupes) = /^\w+:[^:]*:${ARGV[0]}:(.*)$/` |

### 3.7.1 Syntaxe normalisée des expressions régulières :

| | |
|---|---|
| **.** | n'importe quel caractère |
| **\|** | exprime l'alternative<br>*pierre\|paul\|jacques* → *pierre ou paul ou jacques* |
| **()** | groupement de caractères<br>(effet secondaire : le groupement de caractères est accessible par la méthode re.group(...) en Python mémorisé dans une variable nommée de $1 à $9 en Perl) |
| **( ? :)** | supprime la mémorisation dans une variable |
| **( ?=)** | les caractères appariés ne sont pas mémorisés dans $& |
| **[]** | constitution d'un ensemble de caractères dans lequel **un** caractère pourra être choisi<br>[a-zA-Z] : **un** des caractères alphabétiques |
| **^** | appariement en début de ligne |
| **$** | appariement en fin de ligne |
| ***** | répétition de 0 à n fois du caractère (ou du groupement de caractères) précédent |
| **+** | répétition de 1 à n fois du caractère (ou du groupement de caractères) précédent |
| **?** | répétition de 0 à 1 fois du caractère (ou du groupement de caractères) précédent |
| **{n}** | répétition n fois du caractère (ou du groupement de caractères) précédent |
| **{n,}** | répétition au moins n fois du caractère (ou du groupement de caractères) précédent |
| **{n,m}** | répétition de n à m fois du caractère (ou du groupement de caractères) précédent<br>moo{3} ↔ moooo, (moo){3} ↔ moomoomoo |
| **\d** | un chiffre, soit un élément de l'ensemble suivant : $[0-9]$ |
| **\w** | un caractère alphanumérique, soit un élément de l'ensemble suivant : [a-zA-Z_0-9] |
| **\w+** | un mot composé des caractères alphanumériques précédents |
| **\W** | un caractère non alphanumérique |
| **\b** | une frontière de mot (entre \w et \W) |
| **\s** | un élément de l'ensemble suivant : [ \t\n\r\f] |
| **\S** | tout élément non compris dans l'ensemble précédent |
| **** | déspécialise le caractère suivant |

### 3.7.2 Remplacement de chaînes de caractères :

En Python :

Les remplacements sont effectués par la méthode **sub** du module **Python re** :
resultat = re.sub(sous-chaîne_à_remplacer, sous-chaîne_de_remplacement, chaîne_cible)

chaine = "Mais ce que j'adore le plus ce sont les anchois"
chaine = re.sub ('anchois', 'anchois marinés à la catalane', chaine)

En Perl :

chaîne_cible =~ **s**/chaîne_à_remplacer/chaîne_de_remplacement/options

**Principales options possibles :**
**g** remplace toutes les occurrences, **i** ne distingue pas les minuscules des majuscules

**Exemples :**

| | |
|---|---|
| s/^([^ ]+) +([^ ]+)/$2 $1/ | inversion des deux premiers mots |
| s/\bmot1\b/mot2/g | remplacement de tous les "mot1" par "mot2" |
| $nombre =~ s/\bmot1\b/mot2/g | remplacement et comptage du nombre de remplacement |

### 3.7.3 Remplacement de caractères (en Perl) :

**tr**/ensemble de caractères à remplacer/ensemble de caractères de remplacement/
**Exemple :**
tr/`[A-Z]`/[a-z]/ remplacement des majuscules par des minuscules

## 3.8 Quelques fonctions prédéfinies de base :

| | en Python | En Perl |
|---|---|---|
| appel d'un exécutable | **os.system("...")** | **system** "..." |
| résultat d'un exécutable | liste = os.popen("...").readlines() | @tableau = '...' |
| affichage sur la sortie standard | **print**(liste_de_valeurs_à_afficher) | **print** liste_de_valeurs_à_afficher |
| découpage suivant un motif | liste = string.split(chaîne, séparateur) | @tableau = **split** /motif de séparation/, chaîne |

## 3.9 Manipulation des répertoires :

En Python :
- récupération du contenu d'un répertoire : **os.listdir(repertoire)**
- test pour savoir si un fichier (au sens large) est un répertoire : **os.path.isdir(fichier)**
- test pour savoir si un fichier (au sens large) est un fichier régulier : **os.path.isfile(fichier)**

Structure récursive d'exploration du contenu d'un répertoire :

```
def parcours (repertoire) :
    print("Je suis dans "+repertoire)
    liste = os.listdir(repertoire)
    for fichier in liste :
        if os.path.isdir(...) :
           ...
        else :
           ...
```

En Perl :
- ouverture d'un répertoire : **opendir**
- récupération du contenu d'un répertoire : **readdir**
- test pour savoir si un fichier (au sens large) est un répertoire : avec le spécificateur **-d**
- test pour savoir si un fichier (au sens large) est un fichier régulier : avec le spécificateur **-f**

Structure récursive d'exploration du contenu d'un répertoire :

```
sub parcours {
    my $repertoire = $_[0];
    print "Je suis dans $repertoire\n";
    if (opendir(my $dh, $repertoire)) {
        my @fichiers = readdir($dh);
        foreach $fichier (@fichiers) {
           if (-d ...) {

           }
           else {
              ...
           }
        }
        closedir $dh;
    }
}
```

## 3.10 Manipulation des fichiers :

En Python :
- Ouverture d'un fichier avec **open** : **fd = open("fichier", "mode d'ouverture")**
- Lecture d'une ligne dans une variable : variable = **fd.read()**
- Lecture de toutes les lignes dans une liste : liste = **fd.readlines()**
- Ecriture d'une ligne : **fd.write(variable)**
- Ecriture de plusieurs lignes : **fd.writelines(liste)**
- Fermeture : **fd.close()**

Ouverture et mémorisation de toutes les lignes d'un fichier dans une liste, puis affichage :

```
lignes = open("essai", "r").readlines()
for ligne in lignes :
    print(ligne, end='')  # on empêche la fonction print de rajouter un retour-chariot après chaque
                          # affichage (il y en a déjà un en fin de chaque ligne du fichier)
```

En Perl :

| | | |
|---|---|---|
| lecture sur l'entrée standard | <> | `$entree = <>` ou `$entree = <STDIN>` |
| | boucle de lecture sur un tube | `while (<STDIN>) {...}` |
| ouverture d'un fichier | **open** descripteur_fichier, nom_fichier | |
| | ouverture en lecture seule | `open FICHIER, "fichier"` |
| | ouverture en écriture | `open FICHIER, ">fichier"` |
| | ouverture en ajout | `open FICHIER, ">>fichier"` |
| | ouverture en lecture écriture | `open FICHIER, "+<fichier"` |
| | en lecture d'une commande | `open GROUPES, "ypcat group\|";` |
| lecture d'une ligne du fichier | <descripteur_fichier> | |
| écriture dans le fichier | **print** descripteur_fichier liste | |
| fermeture d'un fichier | **close** descripteur_fichier | |

La variable **$/** détermine la chaîne de caractères marquant les fins de ligne lors de la lecture d'un fichier.
Elle peut être indéfinie pour lire le fichier en une seule fois : `undef $/; $_ = <FICH>;`
ou par exemple, pour s'arrêter sur les points d'un fichier texte : `$/="."; while (<FICH>) {...};`

### 3.11 Comparatif Python/Perl/Java :

Script d'affichage de tous les verbes d'un dictionnaire bilingue, exemple :
(`abaisser verbe -06 push_down\etre/obj pull_down\etre/obj lower/valeur reduce/tempe'rature`)

En Python :

```
#!/usr/bin/env python3

import re, os

recherche = re.compile(r"^(.*)\s+verbe")
fichier = open(os.environ['HOME']+'/RECHERCHE/FONGUS/ANALYSEUR/dictionnaire_francais_anglais', 'r')
lignes = fichier.readlines()
fichier.close()

for ligne in lignes:
    res = re.search(recherche, ligne)
    if res:
        print(res.group(1))
```

En Perl :

```
#!/usr/bin/env perl

open F, $ENV{HOME}.'/RECHERCHE/FONGUS/ANALYSEUR/dictionnaire_francais_anglais';
@lignes = <F>;
close F;
for (@lignes) {
    if (/^(.*)\s+verbe/o) {
        print "$1\n";
    }
}
```

En (vieux) Java :

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class test_re {
  public static void main(String[] params) throws java.io.IOException {
     Pattern pattern = Pattern.compile("^(.*)\\s+verbe");

     String filename = "/auto/pompidor/RECHERCHE/FONGUS/ANALYSEUR/dictionnaire_francais_anglais";
     java.io.BufferedReader fichier = new java.io.BufferedReader(new java.io.FileReader(filename));

     String ligne;
     Matcher match;
     while ( (ligne = fichier.readLine()) != null) {
         match = pattern.matcher(ligne);
         if(match.find()) {   // ou matches
             System.out.println(match.group(1));
         }
     }
     fichier.close();
  }
}
```

## 3.12 Exemples de scripts :

**Affichage des lignes numérotées du fichier passwd d'un ordinateur isolé :**

En Python :

```
#!/usr/bin/env python3

i = 1
fd = open ('/etc/passwd', 'r')   # ouverture du fichier en lecture
for ligne in fd.readlines() :    # lecture de chaque ligne
    print (i, ' : ', ligne)      # numérotation de chaque ligne
    i+=1
fd.close()
```

En Perl :

```
#!/usr/bin/env perl

open F, '/etc/passwd';           # ouverture du fichier en lecture
while (<F>) {                    # lecture de chaque ligne
    print ++$i, " : ", $_}       # numérotation de chaque ligne
close F
```

**Affichage des logins du fichier passwd géré par NIS et redirigé sur l'entrée standard :**
Exemple d'appel : `ypcat passwd | logins.p[y|l]`

En Python :

```
#!/usr/bin/env python3

import sys, re
recherche = re.compile(r"^([^:]+)")

for ligne in sys.stdin.readlines() :   # lecture de chaque ligne de l'entrée standard
    res = re.search(recherche, ligne)  # application de l'expression régulière sur la ligne
    if res:
       print(res.group(1))             # affichage du login
```

En Perl :

```
#!/usr/bin/env perl

while(<STDIN>) {
   print /^([^:]+)/, "\n" # Les parenthèses permettent l'extraction du champ et donc son affichage
}
```

**Affichage du nom complet (et de l'UID) des utilisateurs dont les logins sont passés en paramètres :**
Exemple d'appel : `ypcat passwd | noms_complets.p[y|l] aubert pompidor`

En Python :

```
#!/usr/bin/env python3

import sys, re

for ligne in sys.stdin.readlines() :        # lecture de chaque ligne de l'entrée standard
    for user in sys.argv :                  # pas très malin, pourquoi ?
        recherche = re.compile(user+":\w+:(\w+):\w+:(\w*) (\w*)")
        res = re.search(recherche, ligne)   # application de l'expression régulière sur la ligne
        if res:
            print ("Nom complet =", res.group(3), res.group(2), "de uid =", res.group(1))
            # affichage du prénom et du nom
```

En Perl :

```
#!/usr/bin/env perl

while(<STDIN>) {
  foreach $user (@ARGV) {
    if (/^${user}:\w+:(\w+):\w+:(\w*) (\w*)/) {   # \w = caractère alphanumérique [a-zA-Z_0-9]
       print "Nom complet = $3 $2 de uid = $1 \n";
       last
    }
  }
}
```

**Affichage des dates et du nombre de connexions de la machine :**

*Exemple : pompidor : 2 Nov (2 fois), 3 Nov (3 fois), 12 Nov (3 fois), 15 Nov (4 fois)*

Exemple d'appel : `last | script_dates_connexions.p[y|l]`

En Python :
Cela sera l'objet d'un TP ....

En Perl :

```
#!/usr/bin/env perl

while (<STDIN>)
{
   /^(\w+)[^A-Z]*\w+\s(\w+)\s+(\w+)/;
   $logins{$1} .= "$3 $2, ";
}

foreach $user (sort keys %logins)
{
   @dates = split /,/, $logins{$user};
   pop @dates;

   undef %dates_uniques;
   foreach $date (@dates) { $dates_uniques{$date}++; }

   print "\n$user : ";
   foreach $date (keys %dates_uniques)
   {
      print "$date($dates_uniques{$date} fois), ";
   }
}
```

# 4 Appels système (partie non vue en cours) :

Cette partie n'intéressera que ceux qui voudraient réécrire un interpréteur de commandes et auraient besoin d'interfacer en C un certain nombre d'appels système offerts par le noyau Linux.

## 4.1 Rappels de quelques fonctions C (nécessaires à la réécriture d'un shell) :

**main** : `int main (int argc, char *argv[], char *arge [])`
argc : nombre de composantes de la commande
éléments de la commande :
argv : tableau de pointeurs sur caractères (fin : pointeur NULL)
nouvel environnement :
arge : tableau de pointeurs sur caractères

**printf** : affichage à l'écran
`printf ("format", variables ...)`

**strcmp** : comparaison de deux chaînes de caractères
`#include <string.h>`
`int strcmp (char *string1, char *string2)`
Compare string1 avec string2

**strcpy** : copie d'une chaîne de caractères dans une autre
`#include <string.h>`
`char* strcpy (char *string1, char *string2)`
Copie string2 dans string1 et retourne string2

**exit** : abandon du programme en cours
exit (int status)
status = 0 si arrêt normal

**getenv** : récupération de la valeur d'une variable d'environnement
`#include <stdlib.h>`
`char* getenv (const char *name)`
Retourne la chaîne d'environnement associée à name ou NULL
ligne_de_chemins = (char*) getenv ("PATH") ;

## 4.2 Appels système à partir d'un programme C :

### 4.2.1 Les primitives de base sur les fichiers :

**access** : Test de l'existence d'un fichier
`#include <unistd.h>`
`access (chemin, X_OK)  (R_OK ou W_OK)`
retourne 0 en cas de succès ou -1 sinon

**creat** : Création de nouveaux fichiers ou réécriture d'anciens
`int creat (char *name, int perms)`
`retourne un descripteur de fichier ou -1`
permission standard : 0755

**open** : Ouverture de fichiers existants
`int open (char *name, int flags, 0)`
retourne un descripteur de fichier ou -1
si `#include<fcntl.h>`, flags = O_RDONLY, O_WRONLY ou O_RDWR

**read** : Lecture dans un fichier
`int n_lus = read (int fd, char *buffer, int n)`

**write** : Ecriture dans un fichier
`int n_écrits = write (int fd, char *buffer, int n)`

**close** : Fermeture d'un fichier
`close (int fd)`

### 4.2.2 La primitive de création de processus :

**fork** : Création dynamique d'un nouveau processus s'exécutant de façon concurrente avec le processus qui l'a créé.
```
#include <unistd.h>
int pid fork ()
```
La valeur de retour est différente dans le processus père et dans le processus fils :
→ 0 dans le processus fils
→ l'identité du processus fils créé dans le processus père
→ -1 si échec

### 4.2.3 Les primitives de synchronisation père/fils :

Tout processus se terminant passe dans l'état zombie dans lequel il reste aussi longtemps que son père n'a pas pris connaissance de sa terminaison.

**wait** : Attente des processus fils
```
#include <sys/types.h>
#include <sys/wait.h>
int pid wait (int *pointeur_status)
```
→ Si le processus possède des fils mais aucun zombie, le processus est bloqué jusqu'à ce que l'un de ses fils devienne zombie.
→ Si le processus possède au moins un fils zombie, le processus renvoie l'identité de l'un de ses fils zombie et si l'adresse de pointeur_status est différente de NULL des informations sont fournies sur la terminaison du processus zombie.

**waitpid** : Attente d'un processus
```
#include <sys/types.h>
#include <sys/wait.h>
int pid waitpid (pid_t pid, int *pointeur_status, int options)
```
Attente bloquante ou non bloquante d'un processus fils particulier.
pid : -1 tout processus fils, ¿0 processus fils particulier
options : 0 bloquant, WNOHANG non bloquant

### 4.2.4 Les primitives de duplication de descripteurs :

Acquisition par un processus d'un nouveau descripteur synonyme d'un descripteur déjà existant.
retour : descripteur synonyme ou -1 en cas d'échec.

**dup** : Duplication de descripteurs
```
#include <unistd.h>
int dup (int desc)
```
Force le plus petit descripteur disponible à devenir un synonyme du descripteur desc

**dup2** : Duplication de descripteurs
```
#include <unistd.h>
int dup2 (int desc1, int desc2)
```
Force le descripteur desc2 à devenir un synonyme du descripteur desc1

### 4.2.5 Les primitives de recouvrement :

**exec..** : Recouvrement de processus
→ retour : -1 en cas d'erreur
→ exec.p : recherche du fichier dans les répertoires dénotés par PATH
→ exec.e : passage d'un nouvel environnement

**execl** : `int execl (const char *chemin_ref, const char *arg, ..., NULL)`
Le fichier de nom ref est chargé et la fonction main correspondante est appelée avec la liste des paramètres suivants

**execv** : `int execv (const char *chemin\_ref, const char *argv [])`

### 4.2.6 Les primitives de communication :

Sous Unix différents types de communications existent :
- (par fichiers),
- par signaux,
- par tubes,

— par tubes nommés,
— par envoi de messages,
— par sémaphores,
— ou par mémoires partagées.

**pipe** : Communication par tube non nommé
```
#include <unistd.h>
int pipe (int p[2])
```
Création de deux descripteurs dans la table des processus
p[0] : descripteur en lecture, p[1] : descripteur en écriture
gestion en mode fifo
retour : 0 réussi, -1 échec

## 4.3 Algorithme de réécriture d'un shell simplifié :

Un shell simplifié doit pouvoir traiter les lignes de commandes suivantes :
`commande_1 < entrée1 | commande ... | commande_n > sortie`

Exemple de l'algorithme traitant simplement deux commandes comme suit :
`grep aa < fichier_entree | wc -l > fichier_sortie`

Lecture d'une ligne de commandes au clavier (**read**)
Création d'un tube (**pipe**)
Création du premier fils (**fork**)
- père : attente (**wait**)
- fils 1 : ouverture du fichier d'entrée (**open**)
- fils 1 : redirection de l'entrée standard sur le fichier d'entrée (**dup2**)
- fils 1 : redirection de la sortie standard sur le tube (**dup2**)
- fils 1 : fermeture du descripteur en lecture du tube (**close**)
- fils 1 : recouvrement (**execvp ou execlp**)

Création du second fils (**fork**)
- père : fermeture des descripteurs du tube (**close**)
- père : attente (**wait**)
- fils 2 : création du fichier de sortie (**creat**)
- fils 2 : redirection de la sortie standard sur le fichier de sortie (**dup2**)
- fils 2 : redirection de l'entrée standard sur le tube (**dup2**)
- fils 2 : fermeture du descripteur en écriture du tube (**close**)
- fils 2 : recouvrement (**execvp ou execlp**)

Illustration de l'effet des appels systèmes **dup2** sur l'exemple précédent :

# Index