Université de Metz

U.F.R. SciF.A.

Nom :_____ **Prénom :**_____

Groupe :_____

T.P. de Système d'Exploitation Unix

M.S.T. Télécom 2eAnnée

Année Universitaire 2004/2005

Y. Morère

Cette page est laissée blanche intentionnellement

Table des matières

Les buts fixés pour cette série de tps sont les suivants :

- ☑ Se familiariser avec un système d'exploitation professionnel et maîtriser les commandes de bases.
- ☑ Écrire des scripts (programmes) afin d'automatiser des tâches.
- ☑ Utiliser des programmes évolués du système afin d'automatiser des actions.
- ☑ Enfin se familiariser avec cet OS multitâches par l'intermédiaire de la gestion de processus grâce aux fonctions systèmes.

Évaluation et notation :

- ☑ Le ou les compte-rendus comporteront les commandes saisies, les résultats obtenus ainsi que les réponses aux questions.

Cette page est laissée blanche intentionnellement

Chapitre 1

Vous serez bientôt des gourous UNIX

1.1 Redirections des entrées/sorties, tubes et filtres

1.1.1 Introduction

Tout processus communique avec l'extérieur par l'intermédiaire de trois fichiers appelés *fichiers standards* :

- Le fichier *entrée standard* sur lequel le processus lit ses données
- Le fichier *sortie standard* sur lequel le processus écrit ses résultats
- le fichier *sortie erreur standard* sur lequel le processus écrit ses messages d'erreurs

Par défaut ces fichiers sont associés au terminal :

- l'*entrée standard* est le clavier
- la *sortie standard* et *sortie erreur* sont l'écran

Il est possible de *rediriger* les entrées/sorties standards d'un processus. On peut leur associer un fichier autre que le terminal.

- *commande* `<` *référence* : redirige l'entrée standard de la *commande* sur le fichier dont on donne la *référence*
- *commande* `>` *référence* : redirige la sortie standard *avec écrasement* du fichier nommé
- *commande* `>>` *référence* : redirige la sortie standard *sans écrasement* du fichier nommé
- *commande* `2>` *référence* : redirige la sortie d'erreur *avec écrasement* du fichier nommé
- *commande* `2>>` *référence* : redirige la sortie d'erreur *sans écrasement* du fichier nommé

1.1.2 Créer un fichier contenant la liste des utilisateurs connectés

Réponse :

Vérification :

1.1.3 Enchaînement de processus

Il est possible, dans une même ligne de commande, de lancer plusieurs commandes qui vont s'exécuter soit *séquentiellement*, soit *concurremment* avec communication entre elle par l'intermédiaire d'une zone mémoire appelée *tube (pipe)*.

1.1.4 Lancement en séquence de plusieurs commandes

commande1 ; *commande2* ou (*commande1* ; *commande2*)
Mettre la date et la liste des utilisateurs connectés dans un fichier `essai`
Réponse :

Vérification :

1.1.5 Communication entre processus

commande1 | *commande2* envoie directement la sortie de la *commande1* vers l'entrée de la *commande2*
Lister les fichiers de votre répertoire et afficher ceux qui contiennent `ess` dans leur nom
Réponse :

Vérification :

Chapitre 2

$SHELL : csh répondit l'echo

2.1 Le Shell

2.1.1 Introduction

Sous Unix, il existe plusieurs *langages de commande*, les plus connus sont les suivants : le Bourne-Shell (`sh`), le C-Shell (`csh`), le Bash (Bourne Again Shell de Linux) (`bash`) et le Korn-Shell (`ksh`).
Dans la suite, on passera sous le shell `bash` par la commande `/bin/bash`.

2.1.2 Les variables du Shell

Le Shell donne à l'utilisateur la possibilité de définir des **variables** qui peuvent être utilisées dans la construction de commandes complexes.
Un certain nombre de variables sont prédéfinies dès le moment où l'utilisateur se loge.

Une variable possède un *nom* et une *valeur*. Son *nom* est une chaîne de caractères commençant par une lettre et composée de lettres, de chiffres et du caractère `_`. Sa *valeur* est une chaîne de caractère quelconque.

2.1.3 Définition de nouvelles variables

Le mécanisme d'*affectation* avec la syntaxe *nom*=*valeur* permet de définir une nouvelle variable et de lui affecter une valeur. La valeur de la variable *nom* est donnée par la chaîne `$nom` ou `${nom}` s'il faut isoler la variables des caractères qui suivent.

Exemple :

```
$ x=gh
$ echo $x
gh
$echo $xijk

$echo ${x}ijk
ghijk
$
```

2.1.3.1 Créer une variable `var` contenant la valeur `etudiant_mst_telecom`

Réponse :

Vérification :

2.1.4 Mécanisme de l'accent grave

Le shell substitue un commande placée entre des accents graves ‘ par la chaîne de caractères qui serait envoyée sur la sortie standard.

Exemple :

```
$ a=‘pwd‘
$ echo $a
/usr/lib
$
```

2.1.4.1 Créer une variable `var2` qui renvoie la date du système lorsque l'on veut afficher son contenu

Réponse :

Vérification :

2.1.5 La commande interne `read`

On peut affecter à une ou plusieurs variables des valeurs lues sur l'entrée standard au moyen de la commande interne `read` avec la syntaxe suivante : `read` $var_1\ var_2 \ldots var_n$.
La commande `read` analyse la ligne lue sur l'entrée standard et affecte les chaînes successives aux différentes variables $var_1\ var_2 \ldots var_n$.

Exemple :

```
$ read nom adresse

lucifer 35, rue de la Gehenne    <-- entre au clavier
$ echo $nom
lucifer
$ echo $adresse
35, rue de la Gehenne
$
```

2.1.5.1 Utiliser la commande `read` afin de créer différentes variables contenant votre adresse

Réponse :

Vérification :

2.1.6 Variables prédéfinies

Les variables prédéfinies par défaut sont les suivantes.

`PS1` a pour valeur le premier caractère de l'invite (prompt).

`PS2` a pour valeur le second caractère de l'invite.

`HOME` a pour valeur la référence absolue du répertoire de l'utilisateur.

`LOGNAME` a pour valeur l'identification de l'utilisateur.

`PATH` est une variable très importante. Sa valeur est une chaîne de caractères indiquant une liste de références de tous les répertoires susceptibles de contenir des commandes utilisées par l'utilisateur.

`IFS` a pour valeur l'ensemble des caractères interprétés comme *séparateurs de chaîne* par le shell.

`TERM` est également une variable importante. Elle indique le type de terminal utilisé.

2.1.7 Les commandes `set` et `unset`

La commande `set` permet d'obtenir la liste des variables de l'environnement et de leurs valeurs. La commande `unset` permet de supprimer une variable.

2.1.7.1 Utiliser la commande `set` pour visualiser les variables prédéfinies

Réponse :

Observations :

2.1.8 Variables exportables

Afin d'ajouter une nouvelle variable à l'environnement shell on utilise la commande `export` ou `setenv`. On peut visualiser les variables à l'environnement shell par la commande `env`.

2.1.9 Exécution d'un fichier de commande

On peut rendre un fichier de commande exécutable de 4 manières :

1. en envoyant la commande `sh fichier`, il suffit dans ce cas que `fichier` soit accessible en lecture.
2. en lançant la commande `. fichier`, il suffit dans ce cas que `fichier` soit accessible en lecture.
3. en envoyant la commande `fichier`, mais là il faut que `fichier` soit accessible en lecture, en écriture et en **exécution**.
4. en envoyant la commande `exec fichier` ou `source fichier`, mais là il faut que `fichier` soit accessible en lecture, en écriture et en **exécution**.

2.1.10 Variables maintenues par le shell

Grâce à ces variables, il est possible, à l'intérieur d'un fichier de commande, de faire référence aux arguments de la ligne de commande.

2.1.10.1 Variables de contrôle

Elles donnent des informations sur les processus en cours.

`$` a pour valeur le numéro de processus shell en cours

`!` a pour valeur le numéro du dernier processus lancé en background

`?` a pour valeur le code de retour de la dernière commande exécutée. 0 si la dernière commande s'est exécutée convenablement, non nulle sinon.

2.1.10.2 Variables de position et paramètres d'un fichier de commande

Tout processus shell maintient une liste de chaînes de caractères que l'on peut connaître par la valeur des variables `*`, `#`,`1`, `2`, `3`, `...`, `9`.

`#` a pour valeur le nombre de chaînes présentes dans la liste

`*` a pour valeur la liste des chaînes de caractères

`i` pour `i=1,...,9` a pour valeur la `i`-ème chaîne de caractères

Remarque : Si le processus shell est crée pour exécuter une commande avec des arguments, alors la variable `0` prend pour valeur le nom de la commande et `*` prend pour valeur la liste des arguments.

2.1.10.3 Utiliser les paramètres positionnels

Saisir dans un fichier le script suivant :

```
echo procedure
echo il y a $# paramètres
echo qui sont [$*]
echo le troisième est $3
echo tous les paramètres sont contenus dans la liste [$@]
echo ce script a comme PID [$$]
```

Rendre le script exécutable et le lancer avec les paramètres `a b c d e f g h`
Réponse :

Observations :

2.1.10.4 La commande `expr`

La commande `expr` considère la suite de ses arguments comme une expression (numérique ou chaîne de caractères). Elle l'évalue et affiche le résultat sur la sortie standard.

```
$ expr 4 + 7
11
$
```

Attention :

1. les différents termes intervenant doivent être séparés par des espaces.
2. si les opérateurs utilisés sont des caractères spéciaux, il doivent être déspécialisés (exemple : `>` doit être écrit `\>`).
3. on peut utiliser les parenthèses `(` et `)` pour regrouper des termes (il faut également les déspécialiser).

Les opérateurs utilisables sont les suivants :

- Le **OU** logique `|` : *expression*$_1$ `|` *expression*$_2$ a pour valeur celle de *expression*$_1$ si *expression*$_1$ n'est pas nulle (chaîne vide ou 0) et sinon pour valeur celle de *expression*$_2$ (ou 0 si *expression*$_2$ est vide).
- Le **ET** logique `&` : *expression*$_1$ `&` *expression*$_2$ a pour valeur celle de *expression*$_1$ si *expression*$_1$ et *expression*$_2$ sont toutes 2 non nulles et non vides ; vaut 0 dans le cas contraire.
- Les opérateurs de **comparaisons** : `<`,`>`,`=`,`>=`,`<=`, `!` (différent de). *expression*$_1$ *opérateur expression*$_2$ vaut 1 si le résultat de la comparaison est vrai, 0 sinon.
- Les opérateurs **additifs** : `+` et `-`
- Les opérateurs **multiplicatifs** : `*` multiplication, `/`division, `%` reste.

Excercice : Écrire le script qui permet de renvoyer la valeur en € d'une valeur entrée en francs.
Réponse :

Observations :

Excercice : Écrire le script `somme` qui permet de faire l'addition des nombres saisis au clavier (0 pour finir).

Réponse :

Cette page est laissée blanche intentionnellement

Chapitre 3

Script, programmation et autres douceurs

3.1 Réalisation de tests

En shell on peut tester le code de retour d'une commande. On sait que tout processus se termine en délivrant un code de retour. Le code de retour du processus est affecté à la variable "`?`". La valeur 0 représente la valeur logique VRAI, et toute autre valeur non nulle la valeur logique FAUX.

Exemple : la commande `ls` délivre un code de retour nul si et seulement si son argument est un fichier du répertoire de travail.

3.2 la commande `test`

Cette commande permet de réaliser des tests sur les chaînes de caractères et sur des fichiers. Deux syntaxes sont possibles :

`test` *expression* ou `[` *expression* `]`

La commande `test` délivre un code de retour 0 si l'expression évaluée est vrai et non nul sinon.

3.2.1 Tests sur les chaînes de caractères

Voici les différents types de test disponibles :

`test -z` *chaîne* est VRAI si et seulement si *chaîne* est la chaîne vide
`test -n` *chaîne* est VRAI si et seulement si *chaîne* n'est pas vide
`test` $chaîne_1$ `=` $chaîne_2$
`test` $chaîne_1$ `!=` $chaîne_2$

3.2.2 Tests sur les chaînes numériques

Une *chaîne numérique* est une suite de chiffres.

`test` $chaîne_1$ `-eq` $chaîne_2$ ("equal")
`test` $chaîne_1$ `-nq` $chaîne_2$ ("not equal")
`test` $chaîne_1$ `-lt` $chaîne_2$ ("less than")
`test` $chaîne_1$ `-le` $chaîne_2$ ("less or equal")
`test` $chaîne_1$ `-gt` $chaîne_2$ ("greater then")
`test` $chaîne_1$ `-ge` $chaîne_2$ ("greater or equal")

3.2.3 Tests sur les fichiers

On peut tester sur un fichier son *type*, les *droits d'accès* de l'utilisateur et le fait que sa *taille* est non nulle.

`test -p` *reference* est vrai ssi *reference* est un tube (nommé).
`test -f` *reference* est vrai ssi *reference* est un fichier ordinaire.

`test -d` *reference* est vrai ssi *reference* est un répertoire.
`test -c` *reference* est vrai ssi *reference* est un fichier spécial *en mode caractère*.
`test -b` *reference* est vrai ssi *reference* est un fichier spécial *en mode bloc*.

3.2.4 Tests sur les droits d'accès

On peut tester si un fichier *reference*, est autorisé en lecture, écriture ou exécution pour le propriétaire du processus.

`test -r` *reference* (autorisation en lecture).
`test -w` *reference* (autorisation en écriture).
`test -x` *reference* (autorisation en exécution).

3.2.5 Test sur la taille

`test -s` *reference* est vrai ssi *reference* est un fichier de taille non nulle.

Excercice : Écrire le script qui permet de vérifier qu'une procédure possède 3 arguments.

Réponse :

3.3 La conditionnelle : `if ... fi`

La structure de contrôle `if ... fi` sa syntaxe est la suivante :

`if` $commande_1$
`then` $commande_2$
`else` $commande_3$
`fi`

$commande_1$ est exécutée ; si son code de retour est vrai (0), $commande_2$ est exécutée et on sort de la structure ; si son code de retour est faux (différent de 0), $commande_3$ est exécutée et on sort de la structure.

Exemple :

```
flore:/users/laih2/ymorere >cat existuser
if grep "$1" /etc/passwd > /dev/null
then
        echo "L'utilisateur $1 a un compte"
else
        echo "L'utilisateur $1 n'existe pas"
fi
flore:/users/laih2/ymorere >chmod a+rx existuser
flore:/users/laih2/ymorere >existuser ymorere
L'utilisateur ymorere a un compte
flore:/users/laih2/ymorere >
```

```
flore:/users/laih2/ymorere >nbarg
il manque des arguments
flore:/users/laih2/ymorere >nbarg toto
le traitement
flore:/users/laih2/ymorere >cat nbarg
if test "$#" -eq 0
then
        echo "il manque des arguments"
        exit 1
else
        echo "le traitement"
        #on fait le traitement
fi
flore:/users/laih2/ymorere >nbarg
il manque des arguments
flore:/users/laih2/ymorere >nbarg toto
le traitement
flore:/users/laih2/ymorere >
```

```
flore:/users/laih2/ymorere >cat existfic
if test -d $1
then
        echo "$1 est un repertoire"
elif test -w $1
then
        echo "$1 autorise l'ajout"
elif test -r $1
then
        echo "$1 est lisible"
else
        echo "$1 autre..."
fi

flore:/users/laih2/ymorere >chmod a+rx existfic
flore:/users/laih2/ymorere >existfic pascal
pascal autre...
flore:/users/laih2/ymorere >existfic existfic
existfic autorise l'ajout
flore:/users/laih2/ymorere >
```

Exercice : écrire un script-shell qui affiche convenablement la contenu d'un fichier selon qu'il s'agisse d'un fichier ordinaire (emploi de `cat`) ou d'un catalogue (emploi de `ls`).

Réponse :

3.4 L'aiguillage : `case ... esac`

La structure de contrôle `case ... esac` permet d'exécuter telle ou telle commande en fonction de la valeur d'une certaine chaîne de caractères :

`case` *chaîne*
motif$_1$`)` *commande*$_1$ `; ;`
motif$_2$`)` *commande*$_2$ `; ;`
`..............`
`esac`

On examine si *chaîne* appartient à l'ensemble d'expressions décrit par le modèle *motif*$_1$; Dans ce cas *commande*$_1$ est exécutée et on sort de la structure ; sinon on examine si *chaîne* satisfait *motif*$_2$ et ainsi de suite. Ainsi la commande associée au premier modèle satisfait est exécutée.

```
flore:/users/laih2/ymorere >cat engtofr
case $1 in
        one)    X=un;;
        two)    X=deux;;
        three)  X=trois;;
        *)      X=$1;;
esac
echo $X
flore:/users/laih2/ymorere >
flore:/users/laih2/ymorere >chmod a+rx engtofr
flore:/users/laih2/ymorere >engtofr one
un
flore:/users/laih2/ymorere >engtofr four
four
flore:/users/laih2/ymorere >
```

Exercice : écrire le script shell *fdate* qui affiche la date en français.
Réponse :

3.5 L'itération bornée : `for ...in ...do ...done`

La structure de contrôle

`for` *variable* `in` *chaîne*$_1$, *chaîne*$_2$ ...
`do` *commande*
`done`

affecte successivement à *variable* les chaînes *chaîne*$_1$, *chaîne*$_2$ ..., en exécutant à chaque fois la *commande*. Une liste de chaînes peut être :

- une liste explicite : `for i in 1 2 blanc noir`
- la valeur d'une variable : `for i $var`
- obtenue comme résultat d'une commande : `for i in ‘ls -al‘`
- le caractère `*` qui représente la liste des noms de fichier du répertoire courant (qui ne commencent pas par `.`) : `for i in *`
- absente ou `$*`, il s'agit de la liste de paramètres de position `$1`, `$2`, etc... : `for i in $*`

```
flore:/users/laih2/ymorere >cat listarg
for arg
do
        echo argument : $arg
done
flore:/users/laih2/ymorere >chmod a+rx listarg
flore:/users/laih2/ymorere >listarg camion voiture vélo
argument : camion
argument : voiture
argument : vélo
flore:/users/laih2/ymorere >
```

```
flore:/users/laih2/ymorere >cat alpha
for consonne in b c d f g
```

```
do
        for voyelle in a e i u o
        do
                echo "$consonne$voyelle\c"
        done
        echo "   "
done
flore:/users/laih2/ymorere >chmod a+rx alpha
flore:/users/laih2/ymorere >alpha
babebibubo
cacecicuco
dadedidudo
fafefifufo
gagegigugo
flore:/users/laih2/ymorere >

flore:/users/laih2/ymorere >cat ex1
VAR="3 2 1 BOUM"
for i in $VAR
do
        echo $i
done
flore:/users/laih2/ymorere >chmod a+rx ex1
flore:/users/laih2/ymorere >ex1
3
2
1
BOUM
flore:/users/laih2/ymorere >

flore:/users/laih2/ymorecat ex2
for fic in `ls`
do
        echo $fic present
done
flore:/users/laih2/ymorere >chmod a+rx ex2
flore:/users/laih2/ymorere >ex2
INBOX present
INBOX~ present
Mail present
Mwm present
Mwm.old present
REMY.MPG present
adt present
adt.c present
alpha present
autosave present
cd present
engtofr present
ess present
essai present
ex1 present
ex2 present
existfic present
existuser present
fic.txt present
fic_copie2.txt present
fic_lien2.txt present
ficlien.txt present
```

```
images present
latex present
listarg present
lmitool present
lyx present
lyx.tar.gz present
mbox present
nbarg present
netscape.ps present
news.ps present
nsmail present
ps present
pwd present
```

Exercice : écrire un script-shell liste tous les fichiers contenus dans les répertoires du `PATH`.
Réponse :

3.6 Les itérations non bornées

3.6.1 L'itération `while...do...done`

La structure de contrôle
`while` *commande*$_1$
`do` *commande*$_2$
`done`
exécute répétitivement *commande*$_1$ puis *commande*$_2$ tant que le code de retour de *commande*$_1$ est vrai sinon on sort de la structure.

```
flore:/users/laih2/ymorere >cat deconnecte
while who | grep "$1" > /dev/null
do
        sleep 2
        echo ",\c"
done
echo "\n$1 n'est plus connecte
flore:/users/laih2/ymorere >
flore:/users/laih2/ymorere >deconnecte ecreuse
,,,,,,,,
flore:/users/laih2/ymorere >
```

Exercice : écrire un script-shell invitant l'utilisateur à répondre au clavier par `oui` ou par `non` et réitérant l'invitation jusqu'à ce qu'elle soit satisfaite.
Réponse :

3.6.2 L'itération `until...do...done`

La structure de contrôle
`until` *commande*$_1$
`do` *commande*$_2$
`done`
exécute répétitivement *commande*$_1$ puis *commande*$_2$ tant que le code de retour de *commande*$_1$ est faux sinon on sort de la structure.

```
flore:/users/laih2/ymorere >cat userloger
until who | grep "$1" > /dev/null
do
        echo ",\c"
        sleep 2
done
echo "\n$1 est arrive\007\007"
flore:/users/laih2/ymorere >chmod a+rx userloger
flore:/users/laih2/ymorere >userloger ymorere

ymorere est arrive
flore:/users/laih2/ymorere >
```

Exercice : écrire un script-shell invitant l'utilisateur à répondre au clavier par `oui` ou par `non` et réitérant l'invitation jusqu'à ce qu'elle soit satisfaite en même temps que l'émission d'un signal sonore.
Réponse :

3.7 Les ruptures de séquence `break` et `continue`

Les commandes `break` et `continue` permettent d'interrompre ou de modifier le déroulement d'un boucle d'itération.

1. La commande `break` fait sortir d'une itération `for`, `while`, `until`. Sous la forme `break` *n* elle permet de sortir de *n* niveaux d'imbrication.
2. La commande `continue` permet de passer au pas suivant de l'itération. Sous la forme `continue` *n* elle permet de sortir de *n-1* niveaux d'imbrication et de passer au pas suivant du *n*-ième niveau.

```
flore:/users/laih2/ymorere >cat ficexist
while :
do
        echo nom de fichier : \c
        read fic
        if test -f "$fic"
```

```
        then
                break
        else
                echo $fic fichier inconnu
        fi
done
#le break branche ici
#suite du traitement
echo "fin de traitement"
flore:/users/laih2/ymorere >chmod a+rx ficexist
flore:/users/laih2/ymorere >ficexist
nom de fichier : c
toto
toto fichier inconnu
nom de fichier : c
ess
fin de traitement
flore:/users/laih2/ymorere >
```

Exercice (`break`) : écrire un script-shell qui affiche les 5 premiers fichiers spéciaux d'un catalogue donné en argument.

Réponse :

Exercice (`continue`) : écrire un fichier de commande `options` qui construit une liste des options apparaissant sur la ligne de commande et l'affiche (une option sera définie par une chaîne de caractères commançant par `-`).

Réponse :

3.8 La fonction `select`

La syntaxe est la suivante :

```
select nom [ in mots ; ]
do
    commandes
done
```

Les `mots` sont décomposés en une liste de mots. Cet ensemble de mots est imprimé sur la sortie standard (écran en général), chacun précédé d'un numéro.

Si `in mots` n'est pas spécifié, `$` est pris par défaut. Ensuite le prompt `PS3` est affiché et le système attend une entrée de la part de l'utilisateur sur l'entrée standard (le clavier en général).

Si ce qui est saisi est l'un des numéros correspondant au `mots` affichés, alors le `mot` est affecté à la variable `nom`. Si la réponse saisie est vide, le menu est affiché de nouveau. Si le caractère `EOF` (touches `ctrl-d`) est saisi la commande `select` se termine. Toute autres saisies que celles contenues dans le menu provoque l'affectation de `nom` à `NULL`.

Les `commandes` sont exécutées après chaque sélection jusqu'à ce qu'une instruction `break` soit rencontrée. Il est donc judicieux de coupler l'instruction `select` avec l'instruction `case`.

Le programme suivant :

```
#!/bin/bash
PS3="Votre choix ? "
select choix in  "Choix A" "Choix B";
  do
    case $REPLY  in
              1) echo "$choix  --> $REPLY";;
              2) echo "$choix  --> $REPLY";;
              *) echo "Vous avez tapé n'importe quoi !";
exit ;;
    esac
done
```

donne la sortie suivante :

```
yann@tuxpowered mst_telecom_2 $ ./select.sh
1) Choix A
2) Choix B
Votre choix ? 1
Choix A  --> 1
Votre choix ? 2
Choix B  --> 2
Votre choix ? 3
Vous avez tapé n'importe quoi !
yann@tuxpowered mst_telecom_2 $
```

3.9 Écriture de script

3.9.1 Utilisation de la fonction `test`

Ecrivez un script qui dit si le paramètre passé est :
– un fichier
– un répertoire
– n'existe pas

– Ecrivez un script qui n'affiche que les répertoires
– Ecrivez un script qui n'affiche que les fichiers
– Ecrivez un script qui donne le nombre de fichiers et de répertoires

3.9.2 Utilisation de la fonction selon que (`case`)

En utilisant la strucure case, écrire un script qui :
– affiche un menu
– demande à l'utilisateur de saisir une option du menu
– affiche à l'utilisateur l'option qu'il a choisi
Exemple de ce qui doit s'afficher à l'écran :

```
********** Menu général ************
<1> Comptabilité
<2> Gestion commerciale
<3> Paie
<9> Quitter
************************************
```

3.9.3 Utilisation de la fonction pour (`for`)

En utilisant la structure for, écrire un programme qui donne les valeurs de y d'une fonction pour les valeurs de x allant de -10 à 10 avec un incrément de 1.
– Réalisez le traitement pour les fonctions y=x, y = x puiss2
– Réécrivez les programmes avec la strucure répéter ... jusqu'à
– Adapter le script afin que les bornes -x, +x soient passées en paramètres au script.
– Modifiez le script de façon à ce que l'on puisse passer en paramètres l'incrément.

3.9.4 Utilisation de la fonction si (`if`)

En utilisant la strucure if, écrire un script qui :
– affiche un menu

– demande à l'utilisateur de saisir une option du menu
– affiche à l'utilisateur l'option qu'il a choisi
Exemple de ce qui doit s'afficher à l'écran :

```
********** Menu général ************
<1> Comptabilité
<2> Gestion commerciale
<3> Paie
<9> Quitter
************************************
```

Vous allez utiliser un fichier dans lequel vous stockerez les informations suivantes :
– premier 3
– deuxième 10
– troisième 25
– quatrième 2
– cinquième 12
Construisez un script qui permet de n'afficher que les enregistrements dont la valeur est supérieure à 10.

3.9.5 Utilisation de la fonction répéter jusqu'à (`until...do...done`)

En utilisant la structure `until...do...done`, écrire un script qui :
– demande à un utilisateur de saisir une commande
– exécute la commande ou affiche un message d'erreur si la commande ne s'est pas déroulée.

– répète cette opération tant que l'utilisateur le désire.
Exemple de ce qui doit s'afficher à l'écran :

```
*********************************************************
Saisisez une commande, commande <Q> pour quitter.
*********************************************************
```

3.9.6 Utilisation de la fonction tant que (`while`)

En utilisant la structure while, écrire un script qui :
Tant que l'utilisateur n'a pas tapé 9
– affiche un menu
– demande à l'utilisateur de saisir une option du menu
– affiche à l'utilisateur le résultat de sa commande
Exemple de ce qui doit s'afficher à l'écran :

```
********** Menu général ******************************
<1> Afficher la date (date)
<2> Afficher le nombre de personnes connectées (who)
<3> Afficher la liste des processus (ps)
<9> Quitter
************************************************************
```

3.9.7 Utilisation de la fonction `select`

Vous allez à l'aide de la fonction select réaliser un menu à 4 options pour un utilisateur. Le script doit boucler tant que l'utilisateur n'a pas choisi de quitter.
- Option 1 : Afficher la liste des utilisateur connectés
- Option 2 : Afficher la liste des processus
- Option 3 : Afficher à l'utilisateur son nom, son UID, son GID, son TTY1
- Option 4 : Terminer

3.9.8 Création de fonction shell

-A- En utilisant les structures que vous connaissez, écrire un script qui affiche la table de multiplication d'un nombre donné en paramètre. Exemple mul 4, donne la table de multiplication de 4. Vous afficherez les résultats pour un multiplicateur allant de 1 à 10. L'affichage de la table de multiplication sera réalisé par une fonction affTABLE().

-B- Modifiez le script afin que les bornes du multiplicateur soient passés en paramètres : exemple mul 3 25 35. On veut la table de multiplication de 3*25 jusqu'à 3*35

-C- Modifier le programme de façon à écrire une calculatrice. L'utilisateur saisi un nombre (par exemple 3). Ensuite il saisira des couples opérateur nombre (exemple + 3). Le programme réalisera les opérations jusqu'à ce que l'utilisateur tape l'opérateur "=" et affichera le résultat final.

3.9.9 Traitement des options de la ligne de commande

Vous utiliserez la fonction `getopts` pour vérifier la saisie de l'utilisateur. Réaliser un script d'archivage qui utilisera les options :
– -a (archive)
– -d (désarchive)
– -c (compresse)
– -x (décompresse)
Le fichier ou le répertoire à archiver sera passé en paramètre : exemple `archive -a -c travail`. Attention `archive -a -d` est invalide.

Remarque Pour archiver vous exploiterez la commande tar (uniquement sur les répertoires car il est inutile d'archiver un fichier). Pour compresser gzip.

3.10 Étude des signaux

Le code ci-dessous permet d'intercepter le signal 2 (arrêt depuis le clavier). Tapez le et analysez son fonctionnement.

```
## Utilisation de trap pour intercepter les signaux
## Utiliser CTRL C
    trap "echo \"trap de l'interruption 2\"" 2
while true
   do
           sleep 2000
           echo "Je suis reveillé"
   done
```

Deuxième exemple de commande qui évite l'arrêt d'un processus.

```
while true; do trap "echo Je suis toujours actif ..." 2 ; done
```

3.10.1 Appel inter-processus et étude de signaux

Ecrire un script "pere" qui aura en charge le déclenchement de deux scripts enfants.

Le script enfant1 écrit sans arrêt sur la console le mot "ping"

Le script enfant2 écrit sans arrêt sur la console le mot "pong"

Le script père fonctionne ainsi :

```
Tant que l'on ne met pas fin au processus pere
1 - Déclenche le processus enfant1
  Attend (wait 10)
  Tue le processus enfant1
2 - Déclenche le processus enfant2
  Attend (wait 10)
  Tue le processus enfant2
Fin de tant que
```

Cette page est laissée blanche intentionnellement

Chapitre 4

Filtrer et traîter les chaînes de caractères

4.1 Les filtres

On appelle un *filtre* toute commande qui lit sur son entrée standard, modifie (éventuellement) les données lues et écrit les résultats sur sa sortie standard. En redirigeant l'entrée standard et la sortie standard sur des fichiers, on peut alors prendre en entrée un fichier source et récupérer en sortie un autre fichier.
On peut combiner les filtres entre eux sur la même ligne de commande. Le plus souvent, ils sont utilisés avec des tubes pour former des commande complexes. Les filtres les plus utilisés sont `cat`, `wc`, `tail`, `tr`, `sort`, `sed` et `grep`.

4.1.1 Les expressions régulières

Certains utilitaires de filtrage comme les commandes `grep`, `ed`, `sed`, `more` utilisent des *expressions régulières* pour décrire des lignes de texte.

`.`	désigne n'importe quel caractère, excepté le passage à la ligne
`[...]`	désigne n'importe quel caractère contenu dans les crochets. On peut désigner plusieurs caractères qui se suivent par un tiret (`-`). Par exemple, `[a-z]` désigne une lettre en minuscule, `[0-9a-zA-Z]` un caractère alphanumérique. Si le caractère `^` se trouve en début des crochets, l'expression désigne n'importe quel caractère qui n'est pas entre crochets.
`^`	désigne un début de ligne lorsqu'il est placé en début d'expression.
`$`	désigne une fin de ligne lorsqu'il est placé en fin d'expression.
`\`	permet de déspécialiser le caractère qui suit.
`*`	désigne aucune ou au moins une occurrence du caractère précédent.

4.1.1.1 Le chapeau `^` et le dollar `$`

Les caractères `^` et `$` définissent une position dans la ligne de texte analysée. Le caractère `^` représente le début de ligne ou d'expression, et le caractère `$` représente la fin de ligne ou d'expression.

Exemple : ainsi pour rechercher dans un fichier, toutes les lignes commençant par la lettre B, on aura :

```
$ grep "^B" fichier
Bonbon
Bouton
```

Exemple : ainsi pour rechercher dans un fichier, toutes les lignes se terminant par le mot "Paris", on aura :

```
$ grep "Paris$" fichier
12h15 Paris
20h30 Paris
```

Exercice : Écrire la commande qui permet de rechercher les lignes blanches d'un fichier pour les compter.

Réponse :

Exercice : Écrire la commande qui permet rechercher dans un fichier, les lignes qui commencent par "Henri" et qui se termine par "Paris".

Réponse :

4.1.1.2 Le point .

Le point "." permet de représenter un caractère quelconque dans une expression.

Exemple : ainsi pour rechercher dans un fichier "`F1`", toutes les lignes contenant une chaîne de 8 caractères se terminant par "`al`" , on aura :

```
$ grep '......al' F1%
```

Exercice : Écrire la commande qui permet rechercher les programmes de `/usr/bin` dont le nom a une longueur de 4 caractères et se terminant par un "`r`". On commencera l'expression par un espace afin de limiter à 4 caractères.

Réponse :

4.1.1.3 Les classes de caractères

Les crochets `[]` permettent de spécifier des classes de caractères recherchés dans un fichier. La recherche donne un résultat lorsqu'un des caractères, au moins, se trouvant entre crochets, est retrouvé dans une ligne de l'expression.

Exemple : ainsi pour rechercher dans un fichier "`fichier`", toutes les lignes contenant au moins un des caractères de l'ensemble `A`, `E`, `F` ou `K`, on aura :

```
$ grep [AEFK] fichier%
```

Il est possible de combiner les diverses possibilités des expressions régulières entre elles. Ainsi pour définir toutes les lignes commençant par un des caractères `A`, `E`, `F` ou `K`, on utilisera l'expression régulière suivante "`^[AEFK]`", respectivement "`[AEFK]$`" pour trouver toutes les lignes qui finissent par ces mêmes lettres.
Les ensembles de caractères peuvent être spécifiés sous la forme d'intervalle. Par exemple `[a-z]` signifie toutes les lettres de `a` à `z`.
De même les classes de caractères peuvent être combinées entre elles. Ainsi il est possible de rechercher toutes les suites de caractères dont le premier est pris dans l'ensemble `A` à `E`, le second égal à `o`, `i` ou `u`, suivi de trois caractères quelconques, suivi d'un `b` ou d'un `c` ou d'une lettre comprise entre `i` et `m` par l'expression suivante : `[A-E][oiu]...[bci-m]`.

Exercice : Écrire la commande qui permet de recherche les lignes ne débutant pas par `S`, `H` ou `J` dans un fichier `fichier`.

Réponse :

Exercice : Écrire la commande qui permet de rechercher les lignes commençant par `P`, suivi de `a` ou `i`, puis deux lettres quelconques, et ayant pour cinquième lettre un `r` ou une `a` dans un fichier `fichier`.

Réponse :

4.1.1.4 Les accolades et les répétitions d'ensembles

Les accolades `{}` offrent la possibilité de spécifier des répétitions. Un nombre entre accolades suivant une expression régulière indique le nombre de répétitions que l'on souhaite appliquer à cette expression. Les accolades doivent être introduite par des antislash `\`. Par exemple pour représenter toutes les suites de caractères commençant par une lettre majuscule, suivie de 5 lettres minuscules et suivie de 2 chiifres entre 0 et 9 nous avons : `[A-Z][a-z]\{5\}[0-9]\{2\}`.
Par exemple, pour rechercher dans un fichier `F1` des suites de 8 lettres commençant par `T` ou `t`, on écrira : `grep '[Tt]\{7\}' F1`.
En règle générale, on précise le nombre minimum et maximum de répétitions `\{p,q\}`. Il existe de plus 3 caractères particuliers pour exprimer des répétitions :
– `*` est équivalent à `\{0,\}` : le minimum est zéro et le maximum est infini.
– `+` est équivalent à `\{1,\}` : le minimum est un et le maximum est infini.
– `?` est équivalent à `\{0,1\}` : l'expression est répétée une fois au plus.

Exercice : Écrire la commande qui permet de rechercher dans un fichier `fichier` des lignes commençant par `B` ou `C` et se terminant par les chiffres `2`, `4` ou `6`.

Réponse :

Exercice : Écrire la commande qui permet de rechercher dans un fichier `fichier` des lignes dont le second caractère est commençant par `r` ou `e` et l'avant dernier caractère est le chiffre `4`.

Réponse :

Exercice : Écrire la commande qui permet de rechercher dans un fichier `fichier` des mots d'au moins 6 lettres commençant par un `P`.

Réponse :

Exercice : Écrire la commande qui permet de rechercher dans un fichier `fichier` toutes les lignes pour lesquelles le mot comporte 5 lettres..

Réponse :

4.1.1.5 La spécification de mot

La spécification de mot dans une expression régulière est réalisée en englobant l'expression recherchée par `\<` et `\>`. Ceci permet de considéré l'expression comme un mot séparé du reste de la ligne par un séparateur.
Par exemple pour chercher le mot `Martin` dans le fichier `fichier` en début de ligne, on aura : `grep '^\<Martin\>'`.

4.1.2 Le filtre identité : `cat`

La syntaxe de `cat` est `cat` [*liste de fichiers*]
`cat` affiche sur la sortie standard ce qui est lu sur l'entrée standard, ou successivement les fichiers de la liste. En redirigeant la sortie standard sur un fichier, il est possible de concaténer un ensemble des fichiers.
L'option `-v` permet de visualiser les caractères non imprimables.

Exemple :

```
$cat chrs
a b e
âzê û î
$cat -v chrs
a ^A b ^B e ^E
M-bzM-j M-{ M-n
```

4.1.3 Ligne, Mot et Caractère : `wc`

Le filtre `wc` compte le nombre de lignes, de mots et de caractères sur l'entrée standard ou dans une liste de fichiers donnés en argument et écrit ces nombres sur la sortie standard.

4.1.4 De la tête à la queue : `head` et `tail`

La syntaxe de `head` est `head` [*-nombre*] [*-liste-de-fichiers*]
`head` écrit sur la sortie standard les *nombre* premières lignes (par défaut les 10 premières) lues sur l'entrée standard ou dans chacun des fichiers donnés en argument.

Exemple :

```
$ head -2 prenoms
David
Francis
$ head -2 prenoms seneque
==> prenoms <==
David
Francis
==> seneque <==
Paucis natus est, qui populum aetatis suae cogitat.
Seneque.
$
```

La syntaxe de `tail` est `tail` [-+*nombre*] [-*liste-de-fichiers*]
`tail` écrit sur la sortie standard les lignes de chaque fichiers donnés en argument et situées à partir de *nombre* lignes comptées à partir du début (respectivement de la fin) si l'on choisi l'option `+` (respectivement l'option `-`). Par défaut *nombre* vaut 10.

Exemple :

```
$cat villes
Paris
Londres
Rome
Jerusalem
$ tail -2 villes
Rome
Jerusalem
$ tail +3 villes
Rome
Jerusalem
$
```

4.1.5 Caractère pour caractère : `tr`

La syntaxe de `tr` est `tr` [*chaîne*$_1$] [*chaîne*$_2$]
Dans l'emploi courant de `tr`, les deux *chaînes* ont le même nombre de lettre et les lettres de *chaîne*$_1$ sont distinctes. L'entrée standard est recopiée sur la sortie standard, chaque lettre de *chaîne*$_1$ est remplacée par la lettre correspondante de *chaîne*$_2$.

Exemple :

```
$ tr "[a-z]" "[A-Z]" < prenoms
DAVID
FRANCIS
PHILIPPE
$
```

Notons que l'option `-d` utilisée dans `tr -d` *chaîne* a pour effet d'éliminer tous les caractères de *chaîne*.

Exemple :

```
$ tr -d "[A-Z]" < prenoms
avid
rancis
hilippe
$
```

4.1.6 Un peu d'ordre : `sort`

La commande `sort` est une commande standard d'Unix qui permet d'ordonner des informations.
Cette commande permet de trier un fichier suivant différents critères, mais elle permet aussi de réaliser la fusion, le tri de plusieurs fichiers en un seul. La commande `sort` lit son entrée sur l'entrée standard et écrit sa sortie sur la sortie standard. Il sera donc nécessaire d'utiliser les redirections.
La syntaxe de la commande `sort` est la suivante :
`sort [options]` *fichiers*
Les options permettent de modifier le comportement ou les clés de tri utilisées par la commande. Les options les plus courantes sont les suivantes :
– `-b` saute les espaces en tête de ligne.
– `-td` utilise la lettre `d` comme séparateur de champs.
– `-d` effectue le tri dans l'ordre lexicographique sans tenir compte des caractères spéciaux et de ponctuation.

– `-f` ignore les différences entre majuscules et minuscules.
– `-M` effectue une comparaison chronologique.
– `-r` tri dans l'ordre inverse.

Ces options constituent des règles de tri qui sont appliquées globalement pour la ou les clés de tri utilisées. La notion de clé de tri implique la notion de champ. Un champ est une suite de caractères délimitée par un séparateur de champs ou un saut le ligne. Le séparateur peut être défini par l'option `-t` comme vu plus haut.
La définition d'une clé de tri se fait par la notation *+p1 -p2*. La valeur *p1* spécifie le début de la clé de tri et la valuer *p2* spécifie la fin de la clé. Si la valeur *p2* est absente, la clé de tri est définie de *p1* jusqu'à la fin de la ligne. Les expressions de *p1* et *p2* sont des quantités de la forme : *m[.n]* ou *m* représente la position du champ et *n* représente le caractère du début du champ. Il est important de remarquer que les champs et les caractères d'un champ sont numérotés à partir de 0. Par exemple la spécification *2.1* correspond au second caractère du troisième champ.

Exemple : Soit le fichier *fitest* suivant :

```
$ cat fitest
Orange      18      124
Banane      23      98
Citron      12      112
Pamplemouse 17      65
Mangue      24      33
Goyave      28      12
Pomme       12      148
Poire       15      122
....
```

Le tri de ce fichier sans aucune option donne le résultat suivant :

```
$ sort fitest
Abricot     23      45
Ananas      18      24
Banane      23      98
Brugnon     22      32
Carotte     33      55
Cerise      23      46
Citron      12      112
Fraise      21      43
Goyave      28      12
Mandarine   15      156
Mangue      24      33
....
```

Si l'on désire trier le fichier sur tous les caractères de la ligne à partir du second champ on aura :

```
$ sort +1 fitest
Citron      12      112
Pomme       12      148
Mandarine   15      156
Poire       15      122
Raisin      17      87
Orange      18      124
Ananas      18      24
Peches      18      33
Fraise      21      43
Brugnon     22      32
Banane      23      98
Carotte     33      55
Abricot     23      45
```

```
Cerise      23      46
....
```

Il est possible de faire le tri sur le troisième champ. Et on remarque que les données ne sont pas triées suivant un critère numérique, mais suivant un critère de type de caractère.

```
$ sort +2 fitest
Citron      12      112
Goyave      28      12
Poire       15      122
Orange      18      124
Pomme       12      148
Mandarine   15      156
Mandarine   15      156
Raisin      17      87
Ananas      18      24
Brugnon     22      32
....
```

Afin de retrouver un ordre de tri numérique, il faut préciser l'option `-n` dans la commande de tri.

```
$ sort -n +2 fitest
Goyave      28      12
Ananas      18      24
Brugnon     22      32
Mangue      24      33
Fraise      21      43
Abricot     23      45
Carotte     33      55
....
```

La commande `sort` permet aussi de trier et de fusionner plusieurs fichiers en un.

```
$head fictest > f1
$tail fictest >f2
$ sort f1 f2
Abricot     23      45
Ananas      18      24
Banane      23      98
Brugnon     22      32
Carotte     33      55
Fraise      21      43
Mangue      24      33
....
```

Exercice : Écrire la commande qui permet de trier le fichier `fichier` où les champs sont séparés pas des " :", sur les champs 3 et 4 en ordre inverse.

Réponse :

4.2 Le filtre-éditeur : `sed`

La commande `sed` est une commande puissante qui peut être considérée comme un filtre et comme un éditeur. Son utilisation la plus courante est de recevoir en entrée successivement chaque ligne d'un fichier, lui faire subir éventuellement des modifications et l'envoyer sur la sortie standard.

4.2.1 Utilisation courante de `sed`

Les commandes les plus utilisées sont les commandes de substitution (`s`) et de suppression (`d`).
La commande la plus simple de modification est la substitution `s` sous la forme :
`sed s/`*motif*`/`*chaine*`/` *fichier*
où *motif* est une expression régulière dont la syntaxe sera précisée. Dans notre cas le *motif* est une simple chaîne de caractères.

Exemple : soit le fichier `toufo` qui contient des erreurs.

```
$cat toufo
pierre qui roule n'a masse pas mousse
dabo dabon dabonnet
2 peste soit des avatars et des avares au cieux

Sacha chassa son chat, Sancho secha ses choux;
```

La commande suivante permet de corrigé la chaîne `n'a masse` en la chaîne `n'amasse`.

```
$sed s/"n'a masse"/"n'amasse"/ toufo > toufo.1
$cat toufo.1
pierre qui roule n'amasse pas mousse
dabo dabon dabonnet
2 peste soit des avatars et des avares au cieux

Sacha chassa son chat, Sancho secha ses choux;
```

Essayons de corriger la seconde ligne.

```
$sed s/dabo/dubo/ toufo.1
$cat toufo.1
pierre qui roule n'amasse pas mousse
dubo dabon dabonnet
2 peste soit des avatars et des avares au cieux

Sacha chassa son chat, Sancho secha ses choux;
```

On remarque que seule la première occurence de `dabo` a été corrigée. Il faut demander explicitement une substitution "globale" en ajoutant le *drapeau* `g`.

```
$sed s/dabo/dubo/g toufo.1 > toufo.2
$cat toufo.2
pierre qui roule n'amasse pas mousse
dubo dubon dubonnet
2 peste soit des avatars et des avares au cieux

Sacha chassa son chat, Sancho secha ses choux;
```

Supprimons maintenant le `2` suivi de l'espace. La commande suivante supprime tout chiffre en début de ligne suivi de zéro ou plusieurs espaces :

```
$sed 's/^[0-9]*[ ]*//' toufo.2 > toutfo.3
$cat toufo.3
pierre qui roule n'amasse pas mousse
dubo dubon dubonnet
peste soit des avatars et des avares au cieux

Sacha chassa son chat, Sancho secha ses choux;
```

Ajoutons maintenant un point-virgule à chaque fin de ligne qui se termine par une lettre.

```
$sed 's/[a-zA-Z]$/&;/' toufo.3 > toutfo.4
$cat toufo.4
pierre qui roule n'amasse pas mousse;
dubo dubon dubonnet;
peste soit des avatars et des avares au cieux;

Sacha chassa son chat, Sancho secha ses choux;
```

Le caractère `&` permet de désigner dans la partie remplacement le motif capté dans la partie initiale.

Exercice : Écrire la commande qui permet de remplacer le point-virgule de la dernière ligne par un point et d'écrire le résultat dans le fichier `toufo.5`.

Réponse :

Exercice : Écrire la commande qui permet de supprimer toutes les lignes vides du fichier `toufo.5`.

Réponse :

Exercice : Écrire la commande qui permet d'insérer 2 espaces en début de chaque ligne. On utilisera le fait que, dans un motif, le point "`.`" est un métacaractère désignant n'importe quel caractère autre que *newline*.

Réponse :

Lorsque de nombreuses commandes `sed` sont nécessaires, il devient intéressant de les réunir dans un fichier et de les faire exécuter grâce à la commande :
`sed -f` *fichier_de_commande fichier*

Exercice : Réunir dans un fichier `script.sed` les commandes déjà utilisée et ajouter celle qui est nécessaire pour compléter la correction du fichier `toutfo`.

Réponse :

4.2.2 Les commandes de `sed`

Outre les commandes de substitution `s` et de suppression `d`, il existe d'autres commandes disponibles.

Ajouter : *[adresse]*`a\` suivi d'un texte sur la ligne suivante. Ajoute le texte après l'*adresse* indiquée.

Exemple :

```
$cat seneque
Paucis natus est, qui populum aetatis suae cogitat.
Seneque.
$cat script1
a\
Il est ne pour peu d'hommes, celui qui n'a en tete\
que les gens de son siècle.
$sed -f script1 seneque
$cat seneque
Paucis natus est, qui populum aetatis suae cogitat.
Seneque.
Il est ne pour peu d'hommes, celui qui n'a en tete\
que les gens de son siècle.
$
```

Insérer : *[adresse]*`i\` suivi d'un texte sur la ligne suivante. Insére le texte après l'*adresse* indiquée.

Exemple :

```
$cat prenoms
David
Francis
Philippe
$cat script2
/Phi/i\
Marcel\
Paul
$sed -f script2 prenoms
$cat prenoms
```

```
David
Francis
Marcel
Paul
Philippe
$
```

Changer : *[adresse]*`c\` suivi d'un texte sur la ligne suivante. Remplace la ligne qui se trouve à l'*adresse* par le texte.

Exemple :

```
$cat script3
/Francis/c\
Francois\
Maurice
$sed -f script3 prenoms
$cat prenoms
David
Francois
Marcel
Paul
Philippe
$
```

Afficher l'espace de travail : `l`
La commande `l` affiche l'*espace de travail* de `sed`, c'est à dire, à chaque étape, le contenu de la ligne sur laquelle vont être effectués les modifications éventuelles. Cette commande permet notamment de visualiser les caractères non imprimables affichés par leurs codes ASCII.

Coder des caractères : `y`
La commande `y` permet de remplacer un caractère par un autre :
`y/abc/uvw`
remplace `a` par `u`, `b` par `v` et `c` par `w`.

```
$cat Saintex
Car j'ai vu trop souvent le pitie s'egarer.
saint-exupery.
$sed 2y/aeinprstuxy/AEINPRSTUXY Saintex
Car j'ai vu trop souvent le pitie s'egarer.
SAINT-EXUPERY.
$
```

Afficher les numéros de lignes : *[adresse]*`=`
La commande `=` affiche les numéros de lignes qui se trouvent à l'*adresse*.

```
$sed  /p/= toutbon
1
  pierre qui roule n'amasse pas mousse;
  dubo dubon dubonnet;
3
  peste soit des avatars et des avaricieux;
  Sacha chassa son chat, Sancho secha ses choux.
$
```

Suppression de l'envoi sur la sortie standard : `-n`

Sauvegarder l'espace de travail : *[adresse]*`w` *fichier*
La commande `w` sauvegarde dans le *fichier* les lignes qui se trouvent à l'*adresse*.

```
$sed -n '/p/w toutou' toutbon
  pierre qui roule n'amasse pas mousse;
  peste soit des avatars et des avaricieux;
$
```

Quitter : *[adresse]*`q`
Stoppe le traitement d'un fichier dès que l'adresse a été atteinte.

```
$sed /F/q prenoms
David
Francis
$
```

Exercice : Le petit script que vous allez devoir écrire sera fort utile à ceux qui utilisent conjointement Linux et Windows, et qui doivent échanger des fichiers texte entre ces systèmes. En effet Linux indique les fins de ligne avec un seul caractère (`\n`), alors que windows en utilise 2 (`\r\n`). Les caractères (`\r`) sont souvent affichés sous Linux comme des (`^M`). Il vous faut donc écrire le petit script qui va permettre d'enlever les `^M` à chaque fin de ligne dans une série de fichiers.

Réponse :

Chapitre 5

Unix Avancé : `find` `cron` et `make`

5.1 Trouver ses petits : `find`

`find` parcoure récursivement l'arborescence en sélectionnant des fichiers selon des critères de recherche, et exécute des actions sur chaque fichier sélectionné.

5.1.1 Syntaxe générale

`find` *repertoire_de_depart critere_de_recherche action_a_executer*

Exemple : `$ find $HOME -print`
Cette commande va parcourir toute l'arborescence à partir du répertoire de login `$HOME`, va sélectionner tous les fichiers puisqu'il n'y a aucun critère de recherche et va afficher le nom de chaque fichier trouvé (les répertoires aussi).

5.1.2 Les critères de recherche

`-name` ***modele*** sélectionne uniquement les fichiers dont le nom correspond au modèle.
Attention ! Le modèle doit être interprété par la commande `find` et non par le shell, s'il contient des caractères spéciaux pour le shell (par exemple `*`), ceux-ci doivent être protégés.

Exemple : `$ find /usr/utilisateur -name '*.c' -print`
L'option `-name` n'accepte qu'un argument. S'il y en a plusieurs il faut rajouter `-name` pour chacun.
De même il ne faut pas oublier de mettre une action (par exemple `-print`), sinon la recherche s'effectuera mais il ne se passera rien.

`-perm` ***nombre_octal*** sélectionne les fichiers dont les droits d'accès sont ceux indiqués par le nombre octal.

Exemple : `$ find /usr/utilisateur -perm 0777 -print`
affiche tous les fichiers qui sont autorisés en lecture, écriture et exécution pour l'utilisateur propriétaire, les personnes du groupe et tous les autres.

`-type` ***caractere*** sélectionne les fichiers dont le type est celui indiqué par le caractère. C'est à dire :
- `c` pour un fichier spécial en mode caractère,
- `b` pour un fichier spécial en mode bloc,
- `d` pour un répertoire,
- `f` pour un fichier normal,
- `l` pour un lien symbolique.

Exemple : `$ find /usr/utilisateurutilisateur -type d -print`
affiche tous les répertoires et sous-répertoires de `/usr/utilisateur`.

`-links` ***nombre_décimal*** sélectionne les fichiers dont le nombre de liens est donné par le nombre décimal. Si le nombre est précédé d'un `+` (d'un `-`) cela signifie supérieur (inférieur) à ce nombre.

Exemple : `$ find /usr/`*`utilisateur`*` -links +2 -print`
affiche tous les fichiers qui ont plus de 2 liens.

`-user` ***n[ou]m_utilisateur*** sélectionne les fichiers dont l'utilisateur propriétaire est le *nom_utilisateur* ou dont la numéro d'utilisateur (UID) est *num_utilisateur*.

Exemple : `$ find /dev -user `*`utilisateur`*` -print`
affiche tous les fichiers spéciaux qui appartiennent à *`utilisateur`*.

`-size` ***nombre_décimal[c]*** sélectionne les fichiers dont la taille est de *nombre_decimal* blocs. Si on post-fixe le *nombre_decimal* par le caractère `c`, alors la taille sera donnée en nombre de caractère.

`-inum` ***nombre_décimal*** sélectionne les fichiers dont le numéro d'I-node est *nombre_decimal*.

`-atime` ***nombre_décimal*** sélectionne les fichiers qui ont été accèdés dans les *nombre_decimal* derniers jours.

Exemple : `$ find /usr/`*`utilisateur`*` -atime -2 -print`
affiche tous les fichiers qui ont été accèdés dans les 2 derniers jours.

`-mtime` ***nombre_décimal*** sélectionne les fichiers qui ont été modifiés dans les *nombre_decimal* derniers jours.

`-newer` ***fichier*** sélectionne les fichiers qui sont plus récents que celui passé en argument.

5.1.3 Combinaison de critères

Plusieurs critères peuvent être groupés par les opérateurs `(` et `)`. Comme ces deux opérateurs sont des caractères spéciaux, ils doivent être despécialisés.
Si plusieurs critères sont mis à la suite, `find` sélectionne les fichiers qui répondent à tous les critères. Le "ET logique" est donc implicite.

Exemple : `$ find /usr/`*`utilisateur`*` \( -name '*.c' -mtime -3 \) -print`
affiche les fichiers se terminant par "`*.c`" et modifiées dans les 3 derniers jours.
Le "OU logique" est représenté par l'opérateur `-o`.

Exemple : `$ find /usr/`*`utilisateur`*` \( -name '*.txt' -o -name '*.doc' \) -print`
affiche tous les fichiers se terminant par "`*.txt`" ou "`*.doc`".
Le "NON logique" est représenté par l'opérateur `!`.

Exemple : `$ find /usr/`*`utilisateur`*` ! -user `*`utilisateur`*` -print`
affiche tous les fichiers n'appartenant pas à *`utilisateur`*.

5.1.4 Les actions possibles sur les noms de fichiers

`-print` affiche le nom des fichiers sélectionnés sur la sortie standard.

`-exec` ***commande*****`\;`** exécute *commande* sur tous les fichiers sélectionnés. Dans la commande Shell, `{}` sera remplacé par les noms des fichiers sélectionnés.

Exemple : `$ find /usr/`*`utilisateur`*` -name '*.txt' -exec lp {} \;`
envoie à l'impression tous les fichiers se terminant par `.txt` dans l'arborescence `/usr/`*`utilisateur`*.

`-ok` ***commande*** `\ ;` même chose que `-exec`, mais demande confirmation avant chaque exécution.

Exemple : `$ find /usr/c1 -name '*.sav' -ok lp {} \ ;`

```
lp /usr/\textsl{utilisateur}/jour.sav ? y
lp /usr/\textsl{utilisateur}/nuit.sav ? y
lp /usr/\textsl{utilisateur}/essai.sav ? n
lp /usr/\textsl{utilisateur}/toto.sav ? n
```

imprime (ou non) chaque fichier `.sav` dans l'arborescence `/usr/`*`utilisateur`* après avoir demandé confirmation.

Exercice : Dans les deux commandes suivantes, la première fonctionne, mais pas la suivante. Pourquoi ?

```
find /usr/\textsl{utilisateur} \( -name '*.c' -mtime -3 \) -print
find '/usr/\textsl{utilisateur} \( -name *.c -mtime -3 \) -print'
```

Réponse :

Exercice : Dans l'arborescence complète, afficher toutes les informations de tous les fichiers dont vous êtes propriétaire.

Réponse :

5.2 Commandes retardées : `at` et `crontab`

5.2.1 La commande `at`

La commande `at` permet de lancer une commande un jour donné, à une heure donnée. Une fois que cette commande a été exécutée, elle n'existe plus. Pour l'utiliser il suffit de taper `at` puis l'heure :

```
at 12:30
```

déclenchera la commande à 12h30.
La syntaxe des entrées est la suivante :
– `at 12 :30 11/30/01` déclenchera la commande le 30 novembre 2001 à 12h30.
– `at now +1hour` déclenchera la commande dans 1 heure à partir de maintenant.
– `at 00 :00 +2days` pour exécuter la commandes dans 2 jours à minuit.
Mais jusque la, nous n'avons entré aucune commande. Mais lorsque vous tapez `at 12 :30`, vous obtenez l'invite de la commande `at` :

```
$at 12:30
at>ping -c 5 192.168.0.1
at>^D
$
```

Vous entrez donc la commande que vous désirez effectuer et vous obtenez la réponse suivante :

```
job 1 at 2001-11-10 12:30
```

La commande va envoyer un `ping` sur la machine `192.168.0.1` à 12h30. Il est bien sur possible de faire exécuter un script. Ensuite `at` envoie par mail le résultat de cette commande à l'auteur.
Si vous désirez savoir ce qu'il va se passer vous pouvez tester la commande `at -c 1`. Cette option permet de montrer la commande numéro 1.
La commande `atq` permet de lister toutes les commandes `at`, et la commande `atrm` permet de supprimer un des *job* de `at`.

5.2.2 La commande `crontab`

`crontab` est un utilitaire qui permet de programmer des actions régulières la machine.
Un démon nommé `cron` lit le fichier qui se trouve dans le répertoire `/var/spool/cron` (le plus souvent) et exécute les commandes qui s'y trouvent.
Afin de créer ce fichier, on peut utiliser le programme `crontab` avec l'option `-e` qui permet d'éditer le fichier crontab à l'intérieur de `vi`. Il ne reste plus qu'a entrer les commandes en respectant la syntaxe décrite ci-dessous. Après avoir enregistrer votre nouveau fichier, `crontab` vous affiche le message suivant `installing new crontab`. Il est possible de visualiser tous les `crontab` avec l'option `-l`.
La syntaxe `crontab` est la suivante :
`<minute> <heure> <jour_du_mois> <mois> <jour_de_semaine> <commande>`
– `minute` : de 0 à 59,
– `heure` : de 0 à 23,
– `jour_du_mois` : de 1 à 31,
– `mois` : de 1 à 12,
– `jour_de_semaine` : de 0 à 6, 0 étant le dimanche et ainsi de suite,
– `commande` : peut comporter plusieurs commandes.
Les mois et les jours peuvent aussi être donnés avec les abréviations anglaises : jan, feb, ... et mon, tue, ...
De plus, on sépare les jours, les mois par des virgules, donc par exemple, pour exécuter une commande tous les 15 et 30 du mois on aura `15,30`.
Si on sépare par un tiret `-`, il s'agit alors d'un intervalle. Ainsi `15-30` signifie du 15 au 30.
Le `/` permet de spécifier une répétition. Ainsi `*/3` indique toutes les 3 minutes. `*` peut aussi être utiliser pour signifier tous les jours de semaines, tous les mois, toutes les heures.

Exemples :
– `0 1 1 * * commande` signifie que la commande sera exécutée le premier jour du mois à 1 heure.
– `0 1 * * mon commande` signifie que la commande sera exécutée un fois par semaine le lundi à 1 heure.
– `0 1 1,15 * * commande` signifie que la commande sera exécutée tous les 1 et 15 du mois à 1 heure.
– `0 1 1-15 * * commande` signifie que la commande sera exécutée tous les 15 premiers jours du mois à 1 heure.
– `0 1 */5 * * commande` signifie que la commande sera exécutée tous les 5 jours à 1 heure.
– `*/3 * * * * commande` signifie que la commande sera exécutée toutes les 3 minutes.
La commande suivante efface tous les jours, les fichiers présents dans le répertoire `/var/log` vieux de plus de 7 jours.

```
0 1 * * * find /var/log -atime 7 -exec rm -f {} \;
```

5.2.3 Contrôle

Dans le cas de `at` ou `crontab`, il est possible de définir qui a le droit d'utiliser ces commandes. Pour cela il existe les fichiers `/etc/cron.allow` et `/etc/cron.deny` et `/etc/at.allow` et `/etc/at.deny`. Par exemple, pour interdire l'utilisation de la commande `cron` à certains utilisateurs, il suffit d'entrer leurs noms dans le fichier `cron.deny`.

Exercice : Créer un crontab qui sauvegarde toutes les nuits dans le répertoire `/var/sauv` sous la forme d'un fichier `tar` compressé, votre répertoire `home`.

Réponse :

5.3 Automatiser les tâches : `make`

5.3.1 Que fait `make`

Sur les systèmes de la famille Unix, `make` remplit le même rôle que les gestionnaires de projets que l'on retrouve dans la plupart des environnements de développement intégrés (IDE) soit sous windows ou encore MacOS. Quel que soit son nom et sa forme, son objectif est toujours le même : centraliser l'ensemble des fichiers et ressources dont se compose un projet, gérer les dépendances et assurer une compilation correcte.
Ainsi si l'on modifie l'un des fichiers sources, le gestionnaire de projet en tiendra compte et saura qu'il faut recompiler ce fichier et procéder de nouveau à une édition de liens pour obtenir un exécutable à jour.
De plus `make` constitue un puissant langage de programmation, spécialisé dans la gestion des projets.

5.3.2 Dans le vif du sujet

Partons d'un exemple : `helloworld`. Mais dans notre cas il sera décomposé sur 2 fichiers sources et un fichier d'entête.
Fichier `main.c`

```
#include <stdio.h>
#include "helloworld.h"
int main(int argc, char *argv[])
{
    hello();
    exit(0);
}
```

Fichier `helloworld.h`

```
void    hello();
```

Fichier `helloworld.c`

```
#include <stdio.h>
void hello()
{
    printf("bonjour le monde\n");
}
```

Afin de compiler ce programme il est possible de le faire de trois manières différentes :

1. `gcc helloworld.c main.c -o helloworld`
2. `gcc -c helloworld.c`
   `gcc -c main.c`
   `gcc main.o helloworld.o -o helloworld`
3. ou avec `make`

La solution 1 convient bien s'il s'agit d'un petit exemple. Mais s'il est question d'un projet plus important il devient nécessaire d'écrire un script de compilation. Dans ce cas la solution 3, utilisant `make` est la plus élégante, car le programme utilise un fichier `Makefile` qui gère les dépendances entre les fichiers.
Nous allons donc écrire le fichier `Makefile`, il ressemble à ceci :

```
helloworld: main.o helloworld.o
    gcc -o helloworld main.o helloworld.o
main.o: main.c
    gcc -c main.c
helloworld.o: helloworld.c
    gcc -c helloworld.c
```

Un `Makefile` contient ainsi un ensemble de règles, dont chacune est constituée d'une "cible", de "dépendances" et de commandes. Il est important de de réaliser l'indentation des lignes ci-dessus avec des tabulations et non des espaces. Ceci occasionnerait des erreurs lors du `make`

La première règle définit la cible `helloworld` ce qui signifie que son rôle réside dans la production d'un fichier `helloworld`. Cette règle possède 2 dépendances `main.o` et `helloworld.o`. Cela indique que pour élaborer le programme `helloworld`, il faut préalablement disposer de ces 2 fichiers. Il vient ensuite la commande shell qui permet de générer `helloworld` à partir des dépendances. Cette commande consiste à appeler la compilation pour obtenir l'exécutable `helloworld` à partir des deux fichiers objets.
La règle suivante est encore plus simple, elle donne le moyen de créer le fichier objet `main.o`. La syntaxe d'un `Makefile` se révèle donc assez simple. On peut alors l'utiliser en vue de la recompilation de notre programme simplement e, lançant la commande `make helloworld` ou encore plus simplement `make`, car l'outil prend par défaut la première cible trouvée.
Que va t'il se passer ? `Make` cherchera à générer `helloworld` : pour cela il vérifiera d'abord si les fichiers requis sont disponibles. S'il manque par exemple `main.o`, il appliquera alors la règle pour produire ce fichier et ainsi de suite si `main.o` nécessitait d'autres dépendances. Une fois toutes les dépendances satisfaites, la commande pour produire `helloworld` sera exécutée afin d'obtenir notre fichier exécutable.

Exercice : Écrire les différents fichiers nécessaires à la mise en œvre du petit projet décrit ci-dessus (`main.c`, `helloworld.c`, `helloworld.h`) sans oublier le `Makefile`. Lancer la compilation (debugger si nécessaire). Ensuite modifier le fichier `helloworld.c` en remplaçant la phrase `bonjour le monde\n` par ce que vous voulez. Que remarquez-vous ?

Réponse :

5.3.3 Pourquoi passer par `make`

En fait, le principal intérêt de cet outil réside dans le fait qu'il n'effectue que le strict minimum. Ainsi comme vous l'avez fait précédemment, si seul le fichier `helloworld.c` est modifié, lors de la recompilation du projet, `make` constatera que la date de modification de `helloworld.c` est plus récente que la création du fichier `helloworld.o`, donc il le recompilera, par contre dans le cas de `main.c` tout est correct, et il n'a pas besoin de regénérer la fichier objet.
On gagne ainsi un temps considérable lors de la compilation de gros projet, en ne recompilant que ce qui est nécessaire.

5.3.4 Plus loin avec GNU `Make`

Bien sur si `make` apporte un aide non négligeable poue la compilation de projet, écrire un `Makefile` complet devient très vite agaçant dès que le projet devient important. Heureusement pour nous, tout ceci peut être automatisé.
GNU `make` propose des mécanismes grâce auxquels il peut déduire pratiquement tout seul les règles à appliquer. Comme il s'agit d'un langage, `make` gère les variables dont certaines possèdent une signification particulière. Il est important d'en connaître au moins 8 :
– `CC` définit le compilateur C par défaut,
– `CFLAGS` définit les options à lui transmettre,
– `CXX` et `CXXFLAGS` jouent le même rôle pour le compilateur C++,
– `LIBS` définit les librairies à utiliser pour la compilation,
– `DESTDIR` définit le chemin sur lequel le programme se verra installé un fois compilé,
– `@` et `<` représentent respectivement la cible et la dépendance courante.
De plus GNU `Make` possède des règles prédéfinies : ainsi il sait que par défaut il doit produire un fichier `toto.o` à partir d'un fichier `toto.c` en invoquant le compilateur défini par la variable `CC`, avec les options `CFLAGS`. Ainsi il est possible de simplifier le `Makefile` comme suit :

```
CC = gcc
OBJS = main.o helloworld.o
```

```
helloworld : $(OBJS)
    $(CC) -o $(@) $(OBJS)
```

On donne à `CC` la valeur `gcc` et l'on garde également les dépendances dans la variable `OBJS` afin d'éviter de les entrer manuellement. La seule règle que nous avons, indique comment produire `helloworld` à partir des dépendances définie par la variable `OBJS`. On utilise alors le compilateur indiqué par `CC`. On remarque aussi l'emploi de la variable `@` qui, à tout instant, représente la cible de la règle où elle figure ; dans le cas présent sa valeur est donc `helloworld`. Il ne s'avère plus nécessaire d'indiquer les règles pour produire `main.o` et `helloworld.o`.

Exercice : Écrire le nouveau `Makefile` et tester son bon fonctionnement.
Réponse :

5.3.5 Nouvelles règles prédéfinies

Si les règles prévues dans GNU `make` ne vous suffisent pas, il est possible d'en redéfinir des nouvelles. Il est alors possible de créer un fichier `postscript` à partir d'un fichier `dvi` par exemple lors de la rédaction de document sous LaTeX 2_ε.

```
%.ps: %.dvi
    dvips -ta4 -o $(@) $(<)
```

On exprime ici le faite, que l'on crée un fichier `postscript` à partir d'un fichier portant le même nom avec le suffixe `.dvi` en utilisant la commande `dvips`.

5.3.6 `make all`, installation et nettoyage

Afin de simplifier la compilation des programmes, la règle `all` est très intéressante. Celle-ci lancera la compilation complète de notre programme.

```
all: helloworld
```

Ce permet de plus d'avoir un terme générique pour lancer la compilation complète de n'importe quel projet.
La seconde chose importante est l'installation du programme. Cette nouvelle règle est dépendante de la compilation complète (`all`). Grâce à la commande `make install`, il sera possible de copier l'exécutable dans le sous-répertoire `bin` de `DESTDIR` ainsi que ses diverses ressources dans `DESTDIR/share`, la documentation dans `DESTDIR/doc` et la page de manuel dans `DESTDIR/man`.

```
install: all
    cp helloworld $(DESTDIR)/bin
    mkdir -p $(DESTDIR)/doc/helloworld
    cp manual.ps $(DESTDIR)/doc/helloworld
    cp README $(DESTDIR)/doc/helloworld
    cp helloworld.1 $(DESTDIR)/man/man1
```

Une fois l'installation faite, il ne reste plus qu'a faire le ménage des fichiers temporaires créés lors de la compilation et qui ne servent plus. La règle `clean` détruira ces fichiers. Il suffira de taper `make clean` pour obtenir une arborescence propre.

```
clean:
    rm -f *.o *~
```

Bien sur cette règle de dépend d'aucune autre.

Exercice : Compléter le fichier `makefile` précédent afin de pouvoir compiler, installer (dans le répertoire `bin` que vous aurez créé dans votre répertoire) et nettoyer le projet `helloworld`.

Réponse :

Chapitre 6

Introduction à la programmation AWK

6.1 Traitement de fichier texte : awk

6.1.1 Présentation et syntaxe

awk est une commande très puissante, c'est un langage de programmation a elle tout seule qui permet une recherche de chaînes et l'exécution d'actions sur les lignes sélectionnées. Elle est utile pour récuperer de l'information, générer des rapports, transformer des données entre autres.
Une grande partie de la syntaxe a été empruntée au langage c, d'ailleurs **awk** sont les abréviations de ces 3 créateurs dont k pour Kernighan, un des inventeurs du c.
La syntaxe de **awk** est la suivante :

```
awk [-F] [-v var=valeur] 'programme' fichier
```

ou

```
awk [-F] [-v var=valeur] -f fichier-config fichier
```

L'argument **-F** doit être suivi du séparateur de champ (**-F :** pour un " **:**" comme séparateur de champ).
L'argument **-f** suivi du nom du fichier de configuration de awk.
L'argument **-v** définit une variable (**var** dans l'exemple) qui sera utilisée par la suite dans le programme.
Un programme **awk** possède la structure suivante : **critère de sélection d'une chaîne action**, quand il n'y a pas de critère c'est que l'action s'applique à toutes les lignes du fichier.
Exemple :

```
awk -F":" '{print \$NF}' /etc/passwd
```

Il n'y a pas de critères, donc l'action s'applique à toutes les lignes du fichier **/etc/passwd**. L'action consiste à afficher le nombre de champ du fichier. **NF** est une variable prédéfinie d'awk, elle est égale au nombre de champs dans une ligne.
Généralement on utilisera **awk** en utilisant un script.

```
\#!/bin/sh
awk [-F] [-v var=valeur] 'programme' \$1
```

Vous appelerez votre script **mon-script.awk**, lui donnerez des droits en exécution (**755** par exemple), et l'appelerez ainsi :

```
mon-script.awk fichier-a-traiter
```

Dans la suite du cours, on utilisera **awk** en sous entendant que celui-ci est à insérer dans un script.
Le quote **'** se trouve sur un clavier azerty standard avec le 4 et éventuellement l'accolade gauche.
ATTENTION : ils existent plusieurs "variétés" de **awk**, il se pourrait que certaines fonctions ou variables systèmes qui vous sont présentées dans ce cours n'existent pas sur votre UNIX. Faites en sorte si vos scripts awk doivent fonctionner sur des plates-formes différentes d'utiliser **gawk** sous licence GNU qui est totalement POSIX.
J'ai constaté des grosses différences de comportement entre le awk natif qu'on soit sous HP-UX, Solaris et sous LINUX, de même quand on insère la commande dans un script, on fait appel à un shell, suivant son type (bash shell, csh, ksh, ...), vous pouvez avoir quelques surprises.

6.2 Enregistrements et champs

awk scinde les données d'entrée en enregistrements et les enregistrements en champ. Un enregistrement est une chaîne d'entrée délimitée par un retour chariot, un champ est une chaîne délimitée par un espace dans un enregistrement.
Par exemple si le fichier à traiter est **/etc/passwd**, le caractère de séparation étant " **:**", un enregistrement est une ligne du fichier, et un champ correspond au chaîne de caractère séparé par un " **:**" (login :mot de passe crypté :UID :GID :commentaires :home directory :shell).
Dans un enregistrement les champs sont référencés par **$1**, **$2**, ..., **$NF** (dernier champ). Par exemple pour **/etc/passwd $1** correspond au login, **$2** au mot de passe crypté, **$3** à l'UID, et **$NF** (ou **$7**) au shell.
L'enregistrement complet (une ligne d'un fichier) est référencé par **$0**.
Par exemple, si l'on veut voir les champs login et home directory de **/etc/passwd**, on tapera :

```
awk -F":" '{print \$1,\$6}' /etc/passwd
```

6.3 Critères de sélection

6.3.1 Présentation

Un critère peut être une expression régulière, une expression ayant une valeur chaîne de caractères, une expression arithmétique, une combinaison des expressions précédentes.
Le critère est inséré entre les chaînes **BEGIN** et **END**, avec la syntaxe suivante :

```
awk -F":" 'BEGIN{instructions} critères END{instructions}' fichier
```

BEGIN peut être suivi d'instruction comme une ligne de commentaire ou pour définir le séparateur. Exemple **BEGIN print"Vérification d'un fichier"; FS=" :"**. Le texte à afficher peut être un résumé de l'action de **awk**. De même pour **END** on peut avoir **ENDprint "travail terminé"** qui indiquera que la commande a achevé son travail. Le **END** n'est pas obligatoire, de même que le **BEGIN**.

6.3.2 Les expressions régulières

La syntaxe est la suivante :

```
/expression régulière/ {instructions}

$0  /expression régulière/ {instructions}
```

les instructions sont exécutées pour chaque ligne contenant une chaîne satisfaisant à l'expression régulière.

```
expression  /expression régulière/{instructions}
```

les instructions sont exécutées pour chaque ligne où la valeur chaîne de l'expression contient une chaîne satisfaisant à l'expression régulière.

```
expression !/expression régulière/ {instructions}
```

les instructions sont exécutées pour chaque ligne où la valeur chaîne de l'expression ne contient pas une chaîne satisfaisant à l'expression régulière.
Soit le fichier adresse suivant (nom, numéro de téléphone domicile, numéro de portable, numéro quelconque) :

```
gwenael | 0298452223 | 0638431234 | 50
marcel | 0466442312 | 0638453211 | 31
judith | 0154674487 | 0645227937 | 23
```

L'exemple suivant vérifie que dans le fichier le numéro de téléphone domicile (champ 2) et le numéro de portable (champ 3) sont bien des nombres.

```
awk 'BEGIN { print "On vérifie les numéros de téléphone; FS="|"}
   $2 ! /\^[0-9][0-9]*$/ { print "Erreur sur le numéro de
                           téléphone domicile, ligne nř"NR": \n"$0}
   $3 ! /\^[0-9][0-9]*$/ { print "Erreur sur le numéro de
                           téléphone du portable, ligne nř"NR": \n"$0}
END { print "Vérification terminé"} ' adresse
```

BEGIN est suivi d'une instruction d'affichage qui résume la fonction de la commande, et de la définition du séparateur de champ. L'expression **$2** se référe au deuxième champ d'une ligne (enregistrement) de adresse soit le numéro de téléphone domicile, on recherche ceux qui ne contiennent pas de chiffre (négation de contient des chiffres), en cas de succés on affichera un message d'erreur, le numéro de ligne courante, un retour à la ligne, puis le contenu entier de la ligne. L'expression **$3** se référe au troisième champ d'une ligne (enregistrement) de adresse soit le numéro du portable, on recherche ceux qui ne contiennent pas de chiffre (négation de contient des chiffres), en cas de succés on affichera un message d'erreur, le numéro de ligne courante, un retour à la ligne, puis le contenu entier de la ligne. **END** est suivi d'une instruction d'affichage indiquant la fin du travail.

6.3.3 Les expressions relationnelles

Un critère peut contenir des opérateurs de comparaison (**-** <, <=,==, **!**=,>=,>). Exemple avec le fichier adresse suivant :

```
awk 'BEGIN { print "On cherche lignes dont le numéro
                    (champ 4) est supérieur à 30"; FS="|"}
 $4 > 30 { print "Numéro supérieur à 30 à la ligne nř"NR": \n"$0}
END { print "Vérification terminé"} ' adresse
```

6.3.4 Combinaison de critères

Un critère peut être constitué par une combinaison booléenne avec les opérateurs ou (||), et (&&) et non (**!**). Exemple :

```
awk 'BEGIN { print "On cherche la ligne avec judith ou
                    avec un numéro inférieur à 30"; FS="|"}
 $1 = = "judith" || $4 < 30 { print "Personne "$1" numéro
                              "$4" ligne nř"NR": \n"$0}
END { print "Vérification terminé"} ' adresse
```

6.3.5 Plage d'enregistrement délimitées par des critères

La syntaxe est la suivante **critère1,critère2 instructions**. Les instructions sont exécutées pour toute les lignes entre la ligne répondant au critère1 et celle au critère2. L'action est exécutée pour les lignes comprises entre la ligne 2 et 6.

```
awk 'BEGIN NR==2;NR==6 { print "ligne nř"NR":$\backslash$  n"\$0}
END ' adresse
```

6.4 Les actions

6.4.1 Présentation

Les actions permettent de transformer ou de manipuler les données, elles contiennent une ou plusieurs instructions. Les actions peuvent être de différents types : fonctions prédéfinies, fonctions de contrôle, fonctions d'affectation, fonctions d'affichage.
Fonctions prédéfinies traitant des numériques :
atan2(y,x) arctangente de x/y en radian (entre -pi et pi)
cos(x) cosinus (radian)
exp(x) exponentielle à la puissance x
int(x) partie entière
log(x) logarithme naturel

rand(x) nombre aléatoire (entre 0 et 1)
sin(x) sinus (radian)
sqr(t) racine carrée
srand(x) définition d'une valeur de départ pour générer un nombre aléatoire
Fonctions prédéfinies traitant de chaînes de caractères
Pour avoir la liste des fonctions prédéfinies sur votre plate-forme vous devez faire un **man awk**, voici la liste des fonctions les plus courantes sur un système UNIX.
gsub(expression-régulière,nouvelle-chaine,chaine-de-caractères) dans chaine-de-caractères tous les caractères décrits par l'expression régulière sont remplacés par nouvelle-chaine. gsub et équivalent à gensub.
gsub(/a/,"ai","oi") Remplace la chaine oi par ai
index(chaine-de-caractères,caractère-à-rechercher) donne la première occurence du caractère-à-rechercher dans la chaine chaine-de-caractères
n=index("patate","ta") n=3 length(chaine-de-caractères) renvoie la longueur de la chaine-de-caractères
n=length("patate") n=6
match(chaine-de-caractères,expression-régulière) renvoie l'indice de la position de la chaîne chaine-de-caractères, repositionne RSTART et RLENGTH
n=match("PO1235D",/[0-9][0-9]/) n=3, RSTART=3 et RLENGTH=4
printf(format,valeur) permet d'envoyer des affichages (sorties) formatées, la syntaxe est identique de la même fonction en C
printf("La variable i est égale à %7,2f",i) sortie du chiffre i avec 7 caractères (éventuellement caractères vides devant) et 2 chiffres après la virgule.
printf("La ligne est %s",$0) > "fichier.int" Redirection de la sortie vers un fichier avec >, on peut utiliser aussi la redirection ■. Veillez à ne pas oublier les "" autour du nom du fichier.
split(chaine-de-caractères,tableau,séparateur) scinde la chaîne chaine-de-caractères dans un tableau, le séparateur de champ est le troisième argument
n=split("zorro est arrivé",tab," ") tab[1]="zorro", tab[2]="est", tab[3]="arrivé", n=3 correspond au nombre d'éléments dans le tableau
sprintf(format,valeur) printf permet d'afficher à l'écran alors que sprintf renvoie la sortie vers une chaîne de caractères.
machaine=sprintf("J'ai %d patates",i) machaine="J'ai 3 patates" (si i=3)
substr(chaine-de-caractères,pos,long) Extrait une chaine de longueur long dans la chaîne chaine-de-caractères à partir de la position pos et l'affecte à une chaîne.
machaine=substr("Zorro est arrivé",5,3) machaine="o e"
sub(expression-régulière,nouvelle-chaine,chaine-de-caractères) idem que gsub sauf que seul la première occurence est remplacée (gsub=globale sub)
system(chaine-de-caractères) permet de lancer des commandes d'autres programmes
commande=sprintf("ls | grep toto") Exécution de la commande UNIX "ls |grep toto"
system(commande)
tolower(chaine-de-caracteres) retourne la chaîne de caractères convertie en minuscule
toupper(chaine-de-caracteres) retourne la chaîne de caractères convetie en majuscule

6.4.2 Fonctions définies par l'utilisateur

Vous pouvez définir une fonction utilisateur de telle sorte qu'elle puisse être considérée comme une fonction prédéfinie. La syntaxe est la suivant :

```
fonction mafonction(liste des paramètres)
{
  instructions
  return valeur
}
```

6.5 Les variables et opérations sur les variables

6.5.1 Présentation

On trouve les variables système et les variables utilisateurs. Les variables systèmes non modifiables donnent des informations sur le déroulement du programme. Les variables utilisateurs sont définies par

l'utilisateur.

6.5.2 Les variables utilisateur

Le nom des variables est formé de lettres, de chiffres (sauf le premier caractère de la variable), d'underscore. Ce n'est pas nécessaire d'initialiser une variable, par défaut, si c'est un numérique, elle est égale à 0, si c'est une chaîne, elle est égale à une chaîne vide. Une variable peut contenir du texte, puis un chiffre, en fonction de son utilisation **awk** va déterminer son type (numérique ou chaîne).

6.5.3 Les variables prédéfinies (système)

Les variables prédéfinies sont les suivantes (en italique les valeurs par défaut) :
ARGC nombre d'arguments de la ligne de commande néant
ARGIND index du tableau **ARGV** du fichier courant
ARGV tableau des arguments de la ligne de commande néant
CONVFMT format de conversion pour les nombres "%.6g"
ENVIRON tableau contenant les valeurs de l'environnement courant
ERRNO contient une chaîne décrivant une erreur ""
FIELIWIDTHS variable expérimentale à ne pas utiliser
FILENAME nom du fichier d'entrée néant
FNR numéro d'enregistrement dans le fichier courant néant
FS contrôle le séparateur des champs d'entrée " "
IGNORECASE contrôle les expressions régulières et les opérations sur les chaînes de caractères 0
NF nombre de champs dans l'enregistrement courant néant
NR nombre d'enregistrements lus jusqu'alors néant
OFMT format de sortie des nombres (nombre après la virgule) "%.6g"
OFS séparateur des champs de sortie " "
ORS séparateur des enregistrements de sortie
RLENGTH néant longueur de la chaîne sélectionnée par le critère "\ n"
RS contrôle le séparateur des enregistrements d'entrée "\ n"
RSTART début de la chaîne sélectionnée par le critère néant
SUBSEP séparateur d'indiçage "\ 034"

6.5.4 Opérations sur les variables

On peut manipuler les variables et leur faire subir certaines opérations. On trouve différents types d'opérateurs, les opérateurs arithmétiques classiques (+, -, *, /, %(modulo, reste de la division), (puissance)), les opérateurs d'affectation (=, +=, -=, *=, /=, %=, =). Exemples :
var=30 affectation du chiffre 30 à var
var="toto" affectation de la chaîne toto à var
var="toto " "est grand" concaténation des chaînes
"toto " et "est grand", résultat dans var
var=var-valeur var-=valeur expressions équivalentes
var=var+valeur var+=valeur expressions équivalentes
var=var*valeur var*=valeur expressions équivalentes
var=var/valeur var/=valeur expressions équivalentes
var=var%valeur var%=valeur expressions équivalentes
var=var+1 var++ expressions équivalentes
var=var-1 var– expressions équivalentes

6.5.5 Les variables de champ

Comme on l'a déjà vu auparavant les champs d'un enregistrement (ligne) sont désignés par **$1**, **$2**,...**$NF**(dernier champ d'une ligne). L'enregistrement complet (ou ligne) est désigné par **$0**. Une variable champ est a les mêmes propriétés que les autres variables. Le signe **$** peut être suivi par une variable, exemple :

```
i=3
print \$(i+1)
```

Provoque l'affichage du champ 4
Lorsque **$0** est modifié, automatiquement les variables de champs **$1**,..**$NF** sont redéfinies.
Quand l'une des variables de champ est modifée, **$0** est modifié. ATTENTION le séparateur ne sera pas celui définit par **FS** mais celui définit par **OFS** (output field separator). Exemple :

```
awk 'BEGIN{print" # Tous les utilisateurs du groupe users(GID 22)basculeront dans le groupe boulot(G
          ";FS=":";OFS=":"}
    $4 != 22 {print $0} # Si le groupe n'est pas users on fait rien
    $4 ==22 {$4=24;print $0} # Si le groupe est 22, on lui réaffecte 24
END {print"C'est fini"}' /etc/passwd > passwd.essai
```

6.6 Les structures de contrôle

6.6.1 Présentation

Parmi les structures de contrôle, on distingue les décisions (**if**, **else**), les boucles (**while**, **for**, **do-while**, **loop**) et les sauts (**next**, **exit**, **continue**, **break**).

6.6.2 Les décisions (if, else)

La syntaxe est la suivante :

```
if  (condition) si la condition est satisfaite (vraie)
  instruction1 on exécute l'instruction
else sinon
  instruction2 on exécute l'instruction 2
```

Si vous avez une suite d'instructions à exécuter, vous devez les regrouper entre deux accolades. Exemple :

```
awk ' BEGIN{print"test de l'absence de mot de passe";FS=":"}
NF==7
{ #pour toutes les lignes contenant 7 champss
  if ($2=="") # si le deuxième champ est vide (correspond
              # au mot de passe crypté))
  { print $1 " n'a pas de mot de passe"} # on affiche le nom
          # de l'utilisateur ($1=login) qui n'a pas de mot de passe
  else sinon
  { print $1 " a un mot de passe"} # on affiche le nom de
                                   #l'utilisateur possédant un mot de passe
}
END{print"C'est fini") ' /etc/passwd
```

6.6.3 Les boucles (while, for, do-while)

while, **for** et **do-while** sont issus du langage de programmation C. La syntaxe de **while** est la suivante :

```
while  (condition) tant que la condition est satisfaite (vraie)
instruction on exécute l'instruction
```

Exemple :

```
awk ' BEGIN{print"affichage de tous les champs de passwd";FS=":"}
{ i=11 # initialisation du compteur à 1 (on commence par le champ 1)
while(i=<NF) # tant qu'on n'est pas en fin de ligne
  { print $ii # on affiche le champ
  i++ # incrémentation du compteur pour passer d'un champ au suivant
  }
}
END{print"C'est fini") ' /etc/passwd
```

La syntaxe de **for** est la suivante :
for (instruction de départ ; condition ; instruction d'incrémentation) On part d'une instruction de départ, on incrémente instruction on exécute l'instruction jusqu'à que la condition soit satisfaite
Exemple :

```
awk ' BEGIN{print" affichage de tous les champs de passwd";FS=":"}
{
for (i=1;i=><NF;i++) # initialisation du compteur à 1,
    #on incrémente le compteur jusqu'à ce qu'on atteigne
    #NF (fin de la ligne)
  { print $i } # on affiche le champ tant que la
               # condition n'est pas satisfaite  }
END{print"C'est fini") ' /etc/passwd
```

Avec **for** on peut travailler avec des tableaux. Soit le tableau suivant : **tab[1]="patate"**, **tab[2]="courgette"**, **tab[3]="poivron"**. Pour afficher le contenu de chacun des éléments du tableau on écrira :

```
for (index in tab)
{
 print "legume :" tab[index]
}
```

La syntaxe de **do-while** est la suivante :

```
do
{instructions}} on exécute les instructions
while  (condition) jusqu'à que la condition soit satisfaite
```

La différence avec while, est qu'on est sûr que l'instruction est exécutée au moins une fois.

6.6.4 Sauts controlés (break, continue, next, exit)

break permet de sortir d'une boucle, la syntaxe est la suivante :

```
for  (;;;) boucle infinie
{instructions on exécute les instructionss
if  (condition)  break si la condition est satisfaite on sort de la boucle
instructions}
```

Alors que **break** permet de sortir d'une boucle, **continue** force un nouveau passage dans une boucle.
next permet d'interrompre le traitrement sur la ligne courante et de passer à la ligne suivante (enregistrement suivant).
exit permet d'abandonner la commande **awk**, les instructions suivant **END** sont exécutées (s'il y en a).

6.7 Les tableaux

6.7.1 Présentation

Un tableau est une variable se composant d'un certains nombres d'autres variables (chaînes de caractères, numériques,...), rangées en mémoire les unes à la suite des autres. Le tableau est dit unidimensionnelle quand la variable élément de tableau n'est pas elle même un tableau. Dans le cas de tableaux imbriqués on parle de tableau unidimensionnels.
Les termes matrice, vecteur ou table sont équivalents à tableau.

6.7.2 Les tableaux unidimensionnels

Vous pouvez définir un tableau unidimensionnel avec la syntaxe suivante :**tab[index]=variable**, l'index est un numérique (mais pas obligatoirement, voir les tableaux associatifs), la variable peut être soit un numérique, soit une chaîne de caractère. Il n'est pas nécessaire de déclarer un tableau, la valeur initiale des éléments est une chaîne vide ou zéro. Exemple de définition d'un tableau avec une boucle **for**.

```
var=1
for (i=1;i<=NF;i++)
  { mon-tab[i]=var++}
```

On dispose de la fonction delete pour supprimer un tableau (**delete tab**). Pour supprimer un élément de tableau on tapera **delete tab**[**index**].

6.7.3 Les tableaux associatifs

Un tableau associatif est un tableau unidimensionnel, à ceci près que les index sont des chaînes de caractères. Exemple :

```
age["olivier"]=27
age["veronique"]=25
age["benjamin"]=5
age["veronique"]=3
for (nom in age)
 {
print nom " a " age[nom] "ans"
 }
```

On a un tableau **age** avec une chaîne de caractères prénom comme index, on lui affecte comme éléments de tableau un numérique (age de la personne mentionnée dans le prénom). Dans la boucle **for** la variable **nom** est remplie successivement des chaînes de caractères de l'index (**olivier**, **veronique**, ...).
Les valeurs de l'index ne sont pas toujours triées.

6.7.4 Les tableaux multidimensionnels

awk n'est pas prévu pour gérer les tableaux multidimensionnels (tableaux imbriqués, ou à plusieurs index), néanmoins on peut simuler un tableau à deux dimensions de la manière suivante. On utilise pour cela la variable prédéfinie **SUBSEP** qui, rappelons le, contient le séparateur d'indiçage. Le principe repose sur la création de deux indices (**i**, **j**) qu'on va concaténer avec **SUBSEP (i :j)**.

```
SUBSEP=":"
i="A",j="B"
tab[i,j]="Coucou"
```

L'élément de tableau "**Coucou**" est donc indexé par la chaîne "**A :B**".

6.8 Excercices

6.8.1 Numérotation des lignes d'un fichier

Écrire un programme qui prend en entrée un fichier et qui renvoie en sortie le fichier avec les numéros de ligne.

6.8.2 Numérotation cadrée des lignes d'un fichier

En utilisant les variables `FILENAME` et `FNR`, écrire un programme qui numérote les lignes d'un fichier passé en paramètres avec le format suivant :

```
Fichier  hosts
    1 : # un commentaire
    2 : 194.57.140.142  portable.lasc.univ-metz.fr      portable
.....
```

6.8.3 Détection simple de chaine

Écrire un script qui affiche toutes les lignes qui contiennent une chaine de caractère. Cette chaîne ne sera pas passée en paramètres mais écrite dans le script.

6.8.4 Détection par expression régulière

Écrire un script qui affichera les lignes d'un fichier passé en paramètre qui contiennent un `e` puis un `4`.

6.8.5 Un bonjour amélioré

Écrire un programme qui demande votre nom et qui affiche

6.8.6 Table de multiplication

Écrire un script qui effectue la table de multiplication de 5.

6.8.7 Des filtres : Transformer le fichier hosts

Il s'agit d'afficher le fichier `/etc/hosts` d'une manière épurée. On renverra son contenu sous le format suivant :

```
xxx.xxx.xxx.xxx nom_machine_dns alias_machine
```

en enlevant les commentaires.

Chapitre 7

Introduction à la programmation `Perl`

7.1 Introduction à la programmation `Perl`

Larry Wall a inventé Perl pour introduire automatiquement des rapports et des états statistiques dans un système interne de forums de discussions. L'intitulé complet de ce langage reflète d'ailleurs cette première orientation : *Practical Extraction and Report Langage*.
Très vite la structure de Perl, qui se voulait au départ une simple extention de `awk`, évolua sous l'influence de nombreux utilisateurs, pour incorporer des héritages d'autres langages comme `C`, `Sed` ou le `Shell`.

7.2 Généralités

Perl est un langage polyvalent, adaptable à de nombreuses situations, et d'une puissance sans cesse accrue. C'est un langage interprété et il ne convient donc pas pour les applications bas niveau. Par contre pour l'essentiel des tâches logiciels, il convient parfaitement.
Le slogan de la communeauté Perl est "TIMTOWTDI – There Is More Than One Way To Do It". C'est la richesse du langage qui permet ceci.

7.3 Utilisation

L'interpréteur Perl se trouve en général dans le répertoire `/user/bin` est est invoqué sous le nom `perl`. L'invocation de l'interpréteur se fait avec la syntaxe suivante :

```
$ perl [options] -e 'commandes' [arguments]
```

ou :

```
$ perl [options] script [arguments]
```

On peut lancer un script directement avec une ligne `shebang`

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

En pratique, on utilisera toujours l'option `-w` (warning) qui affiche des avertissements justifiés lorsque l'interpréteur rencontre des expressions douteuses dans le code.

```
#!/usr/bin/perl -w
```

ou invoquerons diretement l'interpréteur en ligne de commande avec :

```
$perl -w -e 'commandes'
```

7.4 Expressions et variables

Perl introduit la notion de contexte en matière d'évaluation d'expression. Il en existe 2 : le contexte *scalaire* et le contexte *list*. En fonction du contexte d'évaluation, la même fonction peut fournir des résultats différents.
On dit qu'une expression est évaluée dans un contexte scalaire quand on lui demande de renvoyer une valeur unique. Cette valeur peut être un nombre (entier ou réel) ou une chaîne de caractères.
Dans un contexte de liste, l'évaluation fournit une succession de variables scalaires. Une liste est représentée comme une suite de valeurs scalaires séparées par des virgules, qui est en général encadrée par des parenthèses.
Une expression est évaluée dans le contexte correspondant à ce que l'on attend d'elle. S'il s'agit d'en affecter le résultat à une variable scalaire, elle sera evaluée dans un contexte scalaire. S'il faut l'ajouter à la fin d'une liste, le résultat est demandé dans un contexte de liste.
Par exemple, les constantes `1`, `2`, `3e-4`, `"zéro"` sont des scalaires et `(1,2,3,5,7,11,13,17)` est une liste au même titre que `("alpha","bravo","charlie","delta")` ou `("un",2,"deux",4,"trois")`.
Il y a un autre type de données scalaires, les références. Il s'agit de pointeurs vers des zones mémoire qui permettent d'accéder indirectement au contenu d'un objet, variable ou constante.
Le langage Perl permet de manipuler des variables, structurées sous trois formes distinctes : les variables scalaires, les tableaux classiques et les tables de hachage.

7.4.1 Les variables scalaires

Une variable scalaire de Perl contient une donnée scalaire. N'étant pas typée, la même variable peut contenir successivement une chaîne de caractères, une valeur entière ou réelle, voire une référence qui pointe vers une autre variable. En revanche les opérateurs agissent pour la plupart sur des types de données précis, ce qui conduit l'interpréteur à effectuer des conversion implicites.
Une variable scalaire est toujours préfixée par un caractère `$`. De manière générale, en Perl, toutes les variables sont préfixées d'une lettre qui indique leur type (consultation et affectation).
Exemple de variables scalaires : `scalaires.pl`
Écrire et tester le programme.

```
#! /usr/bin/perl -w

# Affectations de quelques variables

$pi = 3.1415926535;

$pi_sur_2 = $pi / 2;

$mnemo_pi = "Que j'aime a faire apprendre ce nombre utile aux sages";

print $mnemo_pi;
print "\n";
print $pi;
print "\n";
print $pi_sur_2;
print "\n";
```

Quelques remarques :
– l'extention des scripts Perl est `.pl` et celle des modules Perl est `.pm` ;
– les commentaires commencent par le catactère `#` ;
– toutes les instructions doivent se terminer par un point-virgule
Les conversions entre nombre et chaînes sont assurées avec transparence lorsque l'interpréteur en ressent le besoin.

```
yann@yoda:~/script_linux$ perl -e '$chaine="12"; print $chaine+1; print "\n";'
13
yann@yoda:~/script_linux$ perl -e '$chaine="ABCD"; print $chaine+1; print "\n";'
1
yann@yoda:~/script_linux$ perl -e '$chaine="4AB"; print $chaine+1; print "\n";'
```

```
5
yann@yoda:~/script_linux$
```

Les opérateurs et les fonctions du langage Perl agissent toujours sur une type bien déterminé,numérique ou chaîne de caractères, ce qui décide des éventuelles conversions.
Lorsqu'une liste est évaluée dans un contexte scalaire, elle est remplacée par son dernier élément.

```
yann@yoda:~/script_linux$ perl -e '$i=("aze","qsd","wxc"); print $i; print "\n";'
wxc
yann@yoda:~/script_linux$
```

Si nous utilisons l'option `-w`, l'interpréteur Perl nous avertit que les deux premières constantes de la liste ne seront pas utilisées.

```
yann@yoda:~/script_linux$ perl -we '$i=("aze","qsd","wxc"); print $i; print "\n";'
Useless use of a constant in void context at -e line 1.
Useless use of a constant in void context at -e line 1.
wxc
yann@yoda:~/script_linux$
```

Dans certains cas, les éléments intermédiaires pourraient avoir des effets de bords (`$i++`) pendant l'évaluation.
L'évaluation des variables proposées par Perl est très puissante. Supposons que nous comptions une variable scalaire nommée `$i`, qui contienne par exemple le nombre 42. Supposons par ailleurs que nous ayons une variable scalaire nommée `$j` qui contienne la chaîne de caractères `"i"`. Nous pouvons alors écrire :

```
yann@yoda:~/script_linux$ perl -e '$i=42; $j="i"; print $$j; print "\n";'
42
yann@yoda:~/script_linux$
```

Ce concept est appelé *référence symbolique*, par analogie avec les liens symboliques que l'on trouve sur les systèmes Unix. Ici une référence symbolique est une variable qui renferme le nom d'une autre variable.
Lorsque le référence symbolique est complexe (concaténation de deux noms de variables), il est possible d'employer des accolades pour la délimiter explicitement. L'opérateur `'.'` permet de concaténer deux chaînes de caractères ; nous l'utilisons pour regrouper les deux moitiés du nom de la variable `$deux` et accéder à son contenu.

```
yann@yoda:~/script_linux$ perl -e '$deux=2; $i="de"; $j="ux"; print ${$i.$j}; print "\n";'
2
yann@yoda:~/script_linux$
```

On a donc reconstruit dynamiquement le nom de la variable pour en obtenir son contenu.

7.4.2 Tables classique

Les données scalaires peuvent être regroupées dans des tables indexées par une valeur numérique. Une variable table est toujours préfixée par la lettre `'@'` en rappel du `a` de *array*.
Une table peut être initialisée par une constante prenant la forme d'une liste de scalaires entre parenthèses.

```
yann@yoda:~/script_linux$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
print @semaine; print "\n";'
dimlunmarmerjeuvensam
yann@yoda:~/script_linux$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
print $semaine[0]; print "\n";'
dim
yann@yoda:~/script_linux$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
print $semaine[1]; print "\n";'
lun
yann@yoda:~/script_linux$
```

On crée alors une table nommée `@semaine`, dont les éléments sont des chaînes de caractères. Le premier élément est d'indice 0 est `"dim"` et le dernier d'indice 6 est `"sam"`. On peut très bien mélanger au sein de la même table des chaînes et des valeurs numériques. Les éléments sont indépendants.
Si la table est toujours préfixée par `@`, ses éléments individuels sont des scalaires et sont donc préfixés par un `$`.
Pour modifier ou consulter les éléments de la table on utilisera :

```
yann@yoda:~/script_linux$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu",
 "ven", "sam");
print $semaine[1];$semaine[2]="tuesday"; print $semaine [4];
print $semaine [2]; print "\n";'
lunjeutuesday
yann@yoda:~/script_linux$
```

Il est important de comprendre que les espaces des noms de variables scalaires et des tables sont disjointes. La variable `$i` et la table `@i` peuvent coexister sans conflit. De plus il faut comprendre que `$i[0]` est un élément de `@i` et n'a rien a voir avec `$i`.
On peut recopier directement une table dans une autre avec une simple affectation.

```
yann@yoda:~/script_linux$  perl -e '@t=(1,2,3,4); @s=@t;print $s[3];
print "\n";'
4
yann@yoda:~/script_linux$
```

On remarquera que tous les éléments de la table sont dupliqués. Si l'on modifie le contenu de la table initiale, la copie n'est pas modifiée.

```
yann@yoda:~/script_linux$  perl -e '@t=("1","2"); @s=@t;$t[0]="un";
print $s[0]; print " "; print $t[0]; print "\n";'
1 un
yann@yoda:~/script_linux$
```

Un dernière remarque : Lorsque l'on évalue une table dans un contexte scalaire, elle fournit le nombre de ces éléments. Une table est évaluée dans un contexte scalaire quand on essaie de l'affecter dans une variable scalaire.

```
yann@yoda:~/script_linux$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
 $nb_jours=@semaine; print $nb_jours;print "\n";'
7
yann@yoda:~/script_linux$
```

De même

```
yann@yoda:~/script_linux$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
$nb_jours=@semaine; print @semaine + 0;print "\n";'
7
yann@yoda:~/script_linux$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
$nb_jours=@semaine; print @semaine + 2;print "\n";'
9
yann@yoda:~/script_linux$
```

Ici on force l'évaluation de `@semaine` comme une valeur numérique pour l'additionner.
Par contre si l'on a :

```
yann@yoda:~/script_linux$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
print @semaine;print "\n";'
dimlunmarmerjeuvensam
yann@yoda:~/script_linux$
```

La table est évaluée dans un contexte de liste et est remplacée par la liste de tous ses éléments. Ici `print` accole tous les éléments de la liste qu'on lui transmet en argument.
Il faut donc bien distinguer les listes, qui sont une organisation des données, et les tables, qui sont des variables. En particulier l'évaluation d'une liste dans un contexte scalaire en fournit le dernier élément, alors que l'évaluation d'une table dans le même contexte donne le nombre de ses membres.

```
yann@yoda:~/script_linux$ perl -e '@t=("un", "deux", "trois", "quatre"); $i=@t;
print $i;print "\n";'
4
yann@yoda:~/script_linux$ perl -e '$i=("un", "deux", "trois", "quatre"); ;
print $i;print "\n";'
quatre
yann@yoda:~/script_linux$
```

Comme les espaces des noms de variables scalaires et des tables sont disjointes, il est possible de créer une variable scalaire avec le même nom qu'une table. En règle générale on y stockera le nombre d'éléments.

```
yann@yoda:~/script_linux$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
$semaine=@semaine; print $semaine;print "\n";'
7
yann@yoda:~/script_linux$
```

L'indice du dernier élément de la table peut être obtenu en préfixant le nom par `#`. Comme cette valeur est scalaire, elle doit être elle-même préfixée par `$`.

```
[yann@ulysse script_linux]$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
print $#semaine;print "\n";'
6
[yann@ulysse script_linux]$
```

Si l'on utilise un indice négatif pour accéder à un élément, le décompte se fait à rebours à partir de la fin de la table.

```
[yann@ulysse script_linux]$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
print $semaine[-1];print "\n";'
sam
[yann@ulysse script_linux]$
```

Le fait d'accéder à un élément d'indice supérieur au dernier de la table ajoute automatiquement cet élément.

```
[yann@ulysse script_linux]$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
$semaine[7]="sun";print $semaine[7];print "\n";'
sun
[yann@ulysse script_linux]$
```

De même si l'on essaie d'insérer un élément dans le dixième case du tableau les éléments intermédiaires seront automatiquement créés mais vides.

```
[yann@ulysse script_linux]$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
$semaine[7]="sun";$semaine[10]="wed";print $semaine[10];
print "\n";print @semaine+0;print "\n";'
wed
11
[yann@ulysse script_linux]$
```

Le nombre d'élément de la table est donc passé à 11.
On peut accéder à un élément par l'intermédiaire des indices négatifs, mais il n'est pas possible de créer des éléments avant l'indice 0 de la table.
Il est possible d'extraire des sous parties d'une table en fournissant une liste des indices souhaités, séparés par des virgules ou en utilisant le symbole `".."` qui indique un intervalle.

```
[yann@ulysse script_linux]$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
@ouvrable=@semaine[1..5];print @ouvrable;print "\n";'
lunmarmerjeuven
[yann@ulysse script_linux]$
```

```
[yann@ulysse script_linux]$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
@jour_pair=@semaine[0,2,4,6];print @jour_pair;print "\n";'
dimmarjeusam
[yann@ulysse script_linux]$
```

Par contre si vous utilisez l'affectation suivante :

```
@nouvelle=@tab[4];
```

`@tab[4]` ne représente pas l'élément numéro 4 `$tab[4]` mais une table ne contenant qu'un seul élément. Il faudra alors consulter cette valeur par :

```
print $nouvelle[0];
```

Une table ne contient que des scalaires, mais un scalaire peut être une référence pointant vers une table, ce qui permet la création de tableaux multidimensionnels. On peut par exemple écrire :

```
@ouvrable=("lun", "mar", "mer", "jeu", "ven");
@semaine=("dim", @ouvrable, "sam");
```

qui est équivalent à :

```
@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
```

Il faut bien comprendre que le contenu de la table `ouvrable` est intégralement insérée au même niveau que les autres éléments de la liste.
Les listes sont puissantes en Perl. Il est possible de les utiliser en tant que partie gauche d'une affectation. Par exemple :

```
[yann@ulysse script_linux]$  perl -e '($debut,$sommet,$fin)=(4,6,12);
print $debut." ".$sommet." ".$fin."\n";'
4 6 12
[yann@ulysse script_linux]$
```

Ceci permet des choses très interressantes comme les permutations :

```
[yann@ulysse script_linux]$  perl -e '($debut,$sommet,$fin)=(4,6,12);
print "debut:".$debut." fin:".$fin."\n";($debut,$fin)=($fin,$debut);
print "debut:".$debut." fin:".$fin."\n";'
debut:4 fin:12
debut:12 fin:4
[yann@ulysse script_linux]$
```

Perl assure que les valeurs seront correctement échangées ; il réalise les évaluations des expressions de la liste de droite, puis ensuite les assignations.
Lorsque les listes des deux cotés n'ont pas le même nombre d'éléments, Perl agit quand même sans contrainte en ignorant les valeurs surnuméraires dans la liste à gauche du signe égal, ou les éléments en trop dans la liste de droite.
Il est aussi possible de placer dans la liste de gauche une variable tableau.

```
[yann@ulysse script_linux]$  perl -e '($debut,$sommet,$fin)=(4,6,12,15,14);
 print "debut:".$debut." fin:".$fin."\n";($debut,$fin)=($fin,$debut);
print "debut:".$debut." fin:".$fin."\n";'
debut:4 fin:12
debut:12 fin:4
[yann@ulysse script_linux]$
```

```
[yann@ulysse script_linux]$  perl -e '($debut,$fin,@valeurs)=(4,6,12,15,14);
 print "debut:".$debut." fin:".$fin." valeurs:".@valeurs."\n";'
debut:4 fin:6 valeurs:3
[yann@ulysse script_linux]$
```

On remarquera toutefois que le remplissage de la table se fait de manière gloutonne. C'est à dire qu'elle conssomera toutes les valeurs disponibles dans la liste de droite.

```
[yann@ulysse script_linux]$  perl -e '($debut,@valeurs,$fin)=(4,6,12,15,14);
 print "debut:".$debut." fin:".$fin." valeurs:".@valeurs."\n";'
debut:4 fin: valeurs:4
[yann@ulysse script_linux]$
```

On voit ici que le valeur `fin` n'a pas été affectée puisque la table à récupérer les 4 dernières valeurs.
Lorsque l'on connait bien le nombre d'éléments de la liste, il est possible de restreindre l'affectation à des sous-ensembles des listes contenues dans la partie gauche.

```
[yann@ulysse script_linux]$  perl -e '(@vecteur_1[0,1], @vecteur_2[0,1],
 @vecteur_3[0,1])=(1,2,3,4,5,6); print $vecteur_1[0]." ".$vecteur_1[1]."
 ".$vecteur_2[0]." ".$vecteur_2[1]." ".$vecteur_3[0]." ".$vecteur_3[1]."\n";'
1 2 3 4 5 6
[yann@ulysse script_linux]$
```

Ici les différentes tables ne consomment que le nombre d'éléments correspondant à l'intervalle indiqué.
Dans la pratique l'insertion d'une variable table dans une liste qui se trouve en partie gauche d'une affectation se fera presque toujours en dernière position de la liste. Cela est très utile quand on ne connait pas exactement la liste des éléments présents.
On est souvent amené à insérer ou à extraire des éléments en tête ou en fin de liste. Même s'il existe des opérateurs spécialisés, il est possible de les remplacer par de simples affectations :

```
@table=($nouveau,@table);
@table=(@table, $nouveau);
```

permet d'ajouter un élément en début de liste (resp. en fin de liste).

```
($inutile, @table)=@table;
```

permet de supprimer le premier élément de la liste. Pour supprimer le dernier élément de la liste, le méthode la plus simple est de décrémenter l'indice du dernier élément.

```
$#table--;
```

Il est possible d'utiliser la même stratégie pour éliminer le ième élément d'une liste.

```
@table=(@table[0..$i-1], @table[$i+1..$#table]);
```

```
[yann@ulysse script_linux]$ perl -e '@semaine=("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
$i=3; print $i."\n";@semaine=(@semaine[0..$i-1],@semaine[$i+1..$#semaine]);
 print @semaine; print "\n"; '
3
dimlunmarjeuvensam
[yann@ulysse script_linux]$
```

Voici un petit excercice qui va vous permettre de tester tout ce que nous avons vu.

```
#! /usr/bin/perl -w
# fichier tables.pl

@semaine = ("dim", "lun", "mar", "mer", "jeu", "ven", "sam");
for ($i = 0; $i < @semaine; $i ++) {
print "semaine[$i] = $semaine[$i]\n";
}
print "semaine = @semaine\n";

@pairs = @semaine [0, 2, 4, 6];
print "pairs = @pairs\n";

($weekend[0], @ouvrables[0..4], $weekend[1]) = @semaine;
print "weekend = @weekend\n";
print "ouvrables = @ouvrables\n";

($mini, $maxi) = (12, 8);
print "(mini, maxi) = ($mini, $maxi)\n";

if ($mini > $maxi) {
($mini, $maxi) = ($maxi, $mini);
}
print "(mini, maxi) = ($mini, $maxi)\n";
```

Rédigez ce petit script et testez le :

7.4.3 Protection des expressions

D'après le script qui suit et en le testant, expliquer le mécanisme de protection des expressions en Perl :

```
#! /usr/bin/perl -w
#fichier protection.pl

$nombre = 12;
$chaine = "abc def";
@table  = ("un", "deux", "trois");

print "Protection forte avec des apostrophes\n";
print '$nombre $chaine @table \n';

print "\nProtection faible avec des guillemets\n";
print "$nombre $chaine @table \n";

print "\nPas de protection\n";
print 12, $chaine, @table;

print "\n";
```

Ecrivez ce petit script et testez le.

Lorsque l'on désire délimiter explicitement le nom d'une variable, on peut l'encadrer par des accolades. Si l'on souhaite affiche le contenu de la variable `$t`, suivi de la chaîne de caractère `[0]`, sans que cela soit considérer comme le premier élément de la table `@t` on peut ecrire `${t}[0]`

```
[yann@ulysse script_linux]$ perl -e '@t=(1,2); $t=3; print "${t}[0]\n";print "$t[0]\n"'
3[0]
1
[yann@ulysse script_linux]$
```

7.4.4 Tables de hachage

Le troisième type de données important disponible en Perl correspond aux *tables de hachage*. On peut également les trouver sous le nom de tableaux associatifs. Ces tableaux se représentent sensiblement comme des tableaux classiques, mais l'indexation ne se fait plus par nombre entier, mais par une chaîne de caractères que l'on nomme *clé*. Ce type de structure offre un accès efficace aux données.
Le nom de variable d'une table de hachage est préfixée par un `%`. Ses éléments, comme ceux d'une table classique, sont des scalaires et donc préfixés par `$`. Pour accéder à un élément, on place la clé entre accolades. Par exemple si `%semaine` est une table de hachage :

```
$semaine{"lundi"}="monday";
$semaine{"mardi"}="tuesday";
...
```

Et l'on pourra consulter un élément par :

```
print $semaine{"lundi"};
```

qui affichera `monday`.
On remarquera que la chaîne de caractères qui sert de clé peut contenir des caractère accentués, des caractères spéciaux et même des espaces. Lorsque la clé ne contient que des caractères alphanumériques "classiques" (7bits), les guillemets à l'intérieur des accolades sont facultatives.
L'initialisation d'une table de hachage se fait par l'intermédiaire d'une liste qui contient des paires clés/valeurs :

```
%semaine=("lundi", "monday", "mardi", "tuesday", "mercredi", "wednesday",
"jeudi", "thursday", "vendredi", "friday", "samedi", "saturday"),
"dimanche", "sunday");
```

Il est donc possible très rapidement de convertir une table de hachage en table standard par l'expression :

```
@semaine=%semaine;
```

L'implémentation d'une table de hachage ne préserve pas l'ordre de saisie des éléments. Aussi dans la table classique, les paires clés/valeurs ne figureront elles pas dans le même ordre que la liste originale. Le script suivant illustre ceci.

```
[yann@ulysse 14]$ more hachages.pl
#! /usr/bin/perl -w
# fichier hachages.pl

%semaine = ("lundi", "monday", "mardi", "tuesday", "mercredi", "wednesday",
        "jeudi", "thursday", "vendredi", "friday", "samedi", "saturday",
        "dimanche", "sunday");

@semaine=%semaine;

for ($i = 0; $i < @semaine; $i ++) {
print '$semaine [', $i, '] = ', $semaine[$i], "\n";
}
[yann@ulysse 14]$ ./hachages.pl
$semaine [0] = mercredi
$semaine [1] = wednesday
```

```
$semaine [2] = dimanche
$semaine [3] = sunday
$semaine [4] = lundi
$semaine [5] = monday
$semaine [6] = mardi
$semaine [7] = tuesday
$semaine [8] = vendredi
$semaine [9] = friday
$semaine [10] = samedi
$semaine [11] = saturday
$semaine [12] = jeudi
$semaine [13] = thursday
[yann@ulysse 14]$
```

Afin de rendre plus lisible l'association clé/valeur dans les initialisations statiques, le langage Perl propose l'opérateur `=>` :

```
%semaine = (
        "lundi"    => "monday",
        "mardi"    => "tuesday",
        "mercredi" => "wednesday",
        "jeudi"    => "thursday",
        "vendredi" => "friday",
        "samedi"   => "saturday",
        "dimanche" => "sunday"
);
```

Un certain nombre de variables sont prédéfinies en Perl, ce qui permet de paramétrer le comportement de l'interpréteur. Par exemple à son lancement, l'interpréteur définit automatiquement une variable classique `@ARG` qui contient les arguments passés en paramètre du script. Il définit également une table de hachage `%ENV` qui offre un accès aux variables d'environement du processus. Les clés sont les chaînes de caractères des noms des variables. Le script suivant illustre ce que l'on vient d'exposer :

```
#! /usr/bin/perl -w
# fichier getenv.pl

for ($i = 0; $i < @ARGV; $i ++) {
        print "$ARGV[$i] : $ENV{$ARGV[$i]}\n";
}
```

```
[yann@ulysse 14]$ ./getenv.pl USER HOSTNAME HOME INEXISTANTE
USER : yann
HOSTNAME : ulysse.lasc.univ-metz.fr
HOME : /home/yann
Use of uninitialized value in concatenation (.) or string at ./getenv.pl line 3.
INEXISTANTE :
[yann@ulysse 14]$
```

7.5 Les opérateurs

Après avoir vu les différents types de données manipulés par Perl, nous allons examiner les opérateurs qui permettent de jouer sur ces expressions.

7.5.1 Opérateurs Numériques

Nous retrouvons les opérateurs arithmétiques et logiques habituels, ainsi que des opérateurs de manipulation de bits.

Symbole	Nom	Exemple d'utilisation
`or`	OU logique	`if(($a == 0) or ($b == 0)){`
`xor`	OU EXCLUSIF logique	`if(($a == 0) xor ($b == 0)){`
`and`	ET logique	`if(($x < 0) and ($y < 0)){`
`not`	Négation logique	`if(not ($valeur < $mini)){`
`? :`	Test	`$maxi = ($a > $b) ? $a : $b;`
`=`	Affectation	`$i = $j`
`||`	OU logique	`if(($a == 0) \|\| ($b == 0)){`
`&&`	ET logique	`if(($x < 0) && ($y < 0)){`
`|`	OU binaire	`$a = 0x19 | 0x88; # $a contient 0x99`
`^`	OU EXCLUSIF binaire	`$a = 0x37 ^ 0xFC; # $a contient 0xCB`
`&`	ET binaire	`$a = 0x37 & 0xFC; # $a contient 0x34`
`==`	Test d'égalité	`if ($i == $y) {`
`!=`	Test de différence	`if ($i != $y) {`
`<=>`	Comparaison signée	`$a = ($i <=> $y);`
`<`	Test d'infériorité stricte	`if ($i < $y) {`
`<=`	Test d'infériorité	`if ($i <= $y) {`
`>`	Test de supériorité stricte	`if ($i > $y) {`
`>=`	Test de supériorité	`if ($i >= $y) {`
`<<`	Décalage binaire	`$a = 0x12 << 2; # $a contient 0x48`
`>>`	Décalage binaire	`$a = 0xC4 >> 2; # $a contient 0x31`
`+`	Addition	`$a = $x + $y;`
`-`	Soustraction	`$a = $x - $y;`
`*`	Multiplication	`$a = $x * $y;`
`/`	Division	`$a = $x / $y;`
`%`	Modulo	`$s = $t % 60;`
`!`	Négation logique	`$a = ! $resultat;`
`-`	Moins unaire	`$s = -$t;`
`~`	Complementation binaire	`$s = ~ 0x35013501; # $a contient 0x CAFECAFE`
`**`	Exponentiation	`$x = 2 ** $n;`
`++`	Incrémentation	`$i ++;`
`-`	Décrémentation	`$i --;`

L'opérateur de comparaison `<=>` renvoie -1 si son argument de gauche est inférieur à celui de droite, 0 s'ils sont égaux et +1 si l'opérande de gauche est supérieur à celui de droite.
Comme en C, certain opérateur peuvent être combiné avec une opération. On retrouve `+=`, `-=`, `/=`, `*=`.

7.5.2 Opérateurs de chaîne

Le langage Perl ne néglige pas le traitement de chaîne de caractères. Il offre des opérateurs performants. La première opération proposée est la concaténation de chaîne. L'opérateur est un simple `"."`.

```
yann@yoda:~/script_linux$ perl -e '$a="abc" . "def"; print $a . "\n";'
abcdef
yann@yoda:~/script_linux$
```

On dispose aussi d'un opérateur de répétition noté `"x"`, pour rappeler le signe multiplier. Il construit la chaîne en multipliant la chaîne de gauche autant de fois qu'on lui indique à droite.

```
yann@yoda:~/script_linux$ perl -e '$a="abc" x 3 . "def"; print $a . "\n";'
abcabcabcdef
yann@yoda:~/script_linux$
```

Cet opérateur est utile pour faire de la mise en page en mode texte. En guise d'exercice, écrire un petit script qui affiche le texte qu'on lui passe en paramètre en l'encadrant.
Vous devriez avoir un affichage de ce type :

```
yann@yoda:~/script_linux/14$ ./encadre_2.pl toto titi tata tutu
+------------------------------+
| toto                         |
| titi                         |
| tata                         |
| tutu                         |
+------------------------------+
yann@yoda:~/script_linux/14$
```

L'opérateur de répétition peut être utilisé pour initialiser une liste.

```
@table = (1) x 15;
```

est équivalent à

```
@table = (1,1,1,1,1,1,1,1,1,1,1,1,1,1,1);
```

Comme une table évaluée en contexte scalaire renvoie son nombre d'élément, on peut l'utiliser en partie droite de l'opérateur `x`. On peut alors réinitialiser tous les éléments d'une table en faisant :

```
@table = (0) x @table;
```

Il existe des opérateurs de comparaison pour les chaînes de caractères. Ils sont différents des comparateurs numériques et sont représentés sous forme littérale.
L'opérateur `eq` *(equal)* vérifie si deux opérandes sont égaux.

```
if (($chaine eq "oui") or ($chaine eq "non"))
{
...
}
```

Si l'on compare deux chaînes de caractères avec l'opérateur `==`, Perl effectue la conversion des chaînes en valeurs numériques et compare les résultats. D'ou l'utilité de l'option `-w`.

```
yann@yoda:~/script_linux/14$ perl -e '$c="ABC"; $d="DEF"; if ($c == sd) {print "OK\n";}'
OK
yann@yoda:~/script_linux/14$ perl -we '$c="ABC"; $d="DEF"; if ($c == sd) {print "OK\n";}'
Unquoted string "sd" may clash with future reserved word at -e line 1.
Name "main::d" used only once: possible typo at -e line 1.
Argument "sd" isn't numeric in numeric eq (==) at -e line 1.
Argument "ABC" isn't numeric in numeric eq (==) at -e line 1.
OK
yann@yoda:~/script_linux/14$
```

L'opérateur `ne` *(not equal)* teste la différence des chaînes.

```
if (($chaine ne "oui") and ($chaine ne "non"))
{
...
}
```

Les opérateur `gt` *(greater than)* et `lt` *(lesser than)* testent respectivement si une chaîne est supérieure ou inférieure à l'autre. Les comparaisons sont faites caractère par caractère suivant l'ordre des codes Ascii. On dispose également des opérateurs `ge` *(greater or equal)* et `le` *(lesser or equal)* qui sont vrais aussi si les chaînes sont égales. L'opérateur `cmp` se conduit comme `<=>` pour les nombres, en renvoyant les valeurs -1,0 et1 suivant que l'opérande de gauche est supérieur, égal ou inférieur à celui de droite.

7.6 Structures de contrôles

7.6.1 Structure de test

7.6.1.1 Tests avec `if`

Les tests sont réalisés en employant la construction

```
if (condition_1)
{
action_1;
}
elseif
{
action_2;
}
else
{
action_par_defaut;
}
```

La condition évaluée par un test `if` est une expression du type de celles que nous venons de voir. Il faut noter que les parties actions d'une structure `if` doit toujours être encadrées par des accolades, même si elles ne sont composées que d'une ligne.
Un test peu aussi être construit en faisant suivre une instruction simple :

```
action if (condition);
```

l'action n'est réalisée que si la condition est vérifiée. On dit alors que le test est un *modificateur* de l'instruction simple. Par exemple :

```
for ($i=0; $i < @table; $i++)
{
next if ($table[$i]==0);
...
}
```

permet de passer au cas suivant (action `next`) lorsqu'un élément n'est pas intéressant durant le parcours d'une table.

7.6.1.2 Tests avec `unless`

Le mot clé `unless`, qui signifie "sauf si", a le comportement inverse de `if`. Cela signifie qu'une construction

```
unless (condition)
{
action_1;
}
else
{
action_2;
}
```

est équivalente à

```
if (not condition)
{
action_1;
}
else
{
action_2;
}
```

ou encore

```
if (condition)
{
action_2;
}
else
{
action_1;
}
```

La plupart du temps `unless` n'est pas utilisé avec un bloc `else` car le schéma devient difficile à lire. Il est souvent employé comme modificateur d'instructions simples.

7.6.1.3 Tests par court-circuits

Il est possible de rendre dépendantes l'exécution des commandes du shell en les liant par des opérateurs `&&` ou `||`. Le même principe peut être appliqué aux instructions Perl grâce au mécanisme de court circuit. Exemple :

```
expression_1 or expression_2;
```

La préséance du `or` étant très faible, `expression_1` va être entièrement évaluée. Si cette expression est vrais, Perl n'a pas besoin d'aller plus loin et l'`expression_2` est ignorée.
Ceci permet de réaliser des tests *courts-circuits*, que l'on peut lire "*expression_1 doit être vraie sinon essayer expression_2*". Cette structure est très souvent employée avec l'instruction `die` qui permet d'arrêter un script avec une message d'erreur. Voici un exemple qui utilise la fonction `open` pour l'ouverture d'un fichier :

```
if (! open(FIC, $nomp_fichier))
{
die "impossible d'ouvrir $nom_fichier.\n"
}
```

ou encore

```
unless(open(FIC, $nomp_fichier))
{
die "impossible d'ouvrir $nom_fichier.\n"
}
```

ou encore

```
die "impossible d'ouvrir $nom_fichier.\n" unless open(FIC, $nomp_fichier);
```

7.6.2 Structures de boucles

Il y a différentes manières de réaliser des itérations en Perl.

7.6.2.1 boucle `while`

La boucle `while` "tant que" se présente comme ceci :

```
while (condition)
{
action;
}
```

Elle exécute le contenu du bloc *action* tant que la *condition* est vérifiée.
La commande `next` permet de d'abandonner l'itération en cours et de revenir au test. La commande `last` permet de quitter la boucle.
Il est possible d'utiliser l'instruction `while` comme modificateur d'instruction simple :

```
action while(condition);
```

Attention : Si la condition de début est fausse, l'action n'est jamais exécutée.

```
[yann@ulysse yann]$ perl -e 'print "action $i\n" while ($i++ < 4);'
action 1
action 2
action 3
action 4
[yann@ulysse yann]$ perl -e 'print "action $i\n" while (1 == 0);'
[yann@ulysse yann]$
```

Afin que l'action soit exécutée au moins une fois, il faut utiliser le mot clé `do`.
En réalité `do` est une fonction qui exécute le bloc passé en argument et qui renvoie la valeur de la dernière expression évaluée.

```
do
{
action;
} while (condition);
```

exemple :

```
[yann@ulysse yann]$ perl -e 'do {print "action $i\n"} while (1 == 0);'
action
[yann@ulysse yann]$
```

7.6.2.2 Boucle `until`

Le mot clé `until` "jusqu'à ce que" permet de construire des boucles inverses de `while`, l'action étant exécutée tant que la condition n'est pas vérifiée.

```
until '$fin_demandee)
{
$chaine = $lecture_ligne();
traitement_chaine($chaine);
}
```

ou

```
lecture_chaine() until ligne ne "");
```

ou encore

```
do
{
$resultat = decrypte($message, $cle++);
} until (en_clair ($resultat));
```

7.6.2.3 Boucle `for`

La boucle `for` s'emploie comme en C ou avec Awk.

```
for␣(action_initiale;test;action_iterative)
{
action;
}
```

En réalité les trois arguments de la boucle `for` sont des expressions qui sont évaluées :
- la première expression `action_initiale`, est évaluée avant de démarrer la boucle. On l'emploie en général pour fixer la valeur de départ d'un compteur ;
- la deuxième expression `test` est évaluée avant chaque itération de la boucle. Si elle renvoie une valeur fausse, la boucle `for` se termine ;
- la troisième expression est évaluée à la fin de chaque itération. Elle sert en général de compteur.

l'emploi le plus courant est celui-ci :

```
for ($i = 0; $i < @table; i++)
{
print "$i : table[$i] \n";
}
```

Les expressions sont toutes facultatives ; en particulier l'absence de test dans le second membre permet de créer une boucle infinie dont il faudra sortir par une rupture de séquence explicite :

```
for(;;)
{
$touche = menu_principal();
last if ($touche == 'q');
jeu() if ($touche == 'j');
}
```

On peut aussi cumuler plusieurs expressions dans le même membre grâce à l'opérateur ",".

```
for ($i=0,$j=0; $i + $j < $n;$i += $increment_i, $j += $increment_j)
{
...
}
```

La commande `next` permet de passer à l'itération suivante, alors que `last` permet de sortir de la boucle.

7.6.2.4 Boucle `foreach`

Une autre structure sert à itérer automatiquement les éléments d'une liste. Le mot clé `foreach` est un synonyme de `for`. `foreach` s'utilise ainsi :

```
foreach $variable(liste)
{
action;
}
```

La séquence d'action sera répétée en plaçant dans la variable successivement tous les éléments de la liste. Cette dernière peut être fournie sous la forme de constante entre parenthèses et virgules :

```
[yann@ulysse yann]$ perl -e 'foreach $a ("un","deux","trois") {print "$a\n"};'
un
deux
trois
[yann@ulysse yann]$
```

mais cela ne présente que peu d'intérêt. Le plus souvent une liste d'argument sera une variable table :

```
[yann@ulysse yann]$ perl -e '@t=(1,2,3,4,5,6,7,8,9);foreach $a (@t) {print "$a\n"};'
1
2
3
4
5
6
7
8
9
[yann@ulysse yann]$
```

on peut aussi utiliser :

```
for ($i = 0;$i < @t; $i++)
{
print "$t[$i]\n";
}
```

mais ceci est bien moins élégant.
En guise d'exercice, ecrivez ce petit script et testez le. Quelles conclusions tirez-vous.

```
[yann@ulysse 14]$ cat foreach_hachage.pl
#! /usr/bin/perl -w

%semaine = (
        "lundi"    => "monday",
        "mardi"    => "tuesday",
        "mercredi" => "wednesday",
        "jeudi"    => "thursday",
        "vendredi" => "friday",
        "samedi"   => "saturday",
        "dimanche" => "sunday"
);

foreach $jour (%semaine) {
        print "$jour\n";
}
[yann@ulysse 14]$
```

Ecrivez maintenant ce script et testez le. Quelles conclusions tirez-vous.

```
[yann@ulysse 14]$ cat foreach_keys.pl
#! /usr/bin/perl -w

%semaine = (
        "lundi"    => "monday",
        "mardi"    => "tuesday",
        "mercredi" => "wednesday",
        "jeudi"    => "thursday",
        "vendredi" => "friday",
        "samedi"   => "saturday",
        "dimanche" => "sunday"
);

foreach $jour (keys %semaine) {
        print "$jour <=> $semaine{$jour}\n";
}
[yann@ulysse 14]$
```

7.6.2.5 Rupture de séquence

Trois instructions permettent de modifier le comportement des boucles. Il s'agit de `next`, `last` et `redo`. Le mot clé `next` passe à l'expression suivante. Le mot clé `last` fait sortir de la boucle. `redo` possède un comportement légèrement différent. Il reprend l'itération, mais :
- pour les boucles `while`, il ne vérifie pas la condition de boucle ;
- pour les boucles `foreach`, il recommence l'action sans passer à l'élément suivant de la liste ;
- pour les boucles `for`, il n'exécute pas le troisième membre de `for` et ne vérifie pas la condition du second membre.

7.6.2.6 Les étiquettes de bloc

Les boucles `for`, `foreach` et `while/until` peuvent être précédées d'une étiquette. C'est à dire d'un mot -par convention en majuscules- suivi de deux points. Lorsque plusieurs boucles sont imbriquées, les commandes `next`, `last` et `redo` peuvent prendre en argument une étiquette qui indiquent à quelle bouble elle se rapporte. Par défaut l'action de ces arguments s'applique à la boucle la plus interne.

7.7 Définitions de fonctions

7.7.1 Définition et invocation

La définition d'une fonction Perl se fait grâce au mot clé `sub`. Ce dernier doit être suivi du nom de la routine à enregistrer, d'un éventuel prorotype des arguments entre parenthèses, et du bloc représentant la fonction entre accolades.

Comme en C, il existe une différence entre la déclaration d'une fonction et sa définition. Cela signifie que l'on peut déclarer une fonction pour permettre à l'interpréteur de connaître la fonction qu'il aura à utiliser par la suite.

```
sub fonction(arguments);
```

La définition d'une fonction, qui fait aussi office de déclaration s'il n'y en a pas eu se présente ainsi :

```
sub fonction (arguments)
{
instructions;
}
```

Pour invoquer une fonction, il suffit de citer son nom suivi des arguments :

```
fonction $arg1, $arg2;
```

ou

```
fonction ($arg1, $arg2);
```

Mais la seconde est préférable.

7.7.2 Paramètres et résultat

Les paramètres d'une fonction lui sont toujours transmis dans une table simple, nommée `@_`. Cela signifie que le premier argument sera accessible sous le nom `$_[0]`, le deuxième sous le nom `$_[1]` et ainsi de suite jusqu'au dernier `$_[$#_]`.
Une fonction renvoir un résultat par l'intermidiaire de la fonction `return`. Il ne s'agit pas d'un scalaire, mais d'une liste de scalaires.
Ecrivez et testez ce petit programme.

```
[yann@ulysse 14]$ cat exemple_sub_1.pl
#! /usr/bin/perl -w

sub somme
{
        $somme = 0;
        foreach $val (@_) {
                $somme += $val;
        }
        return ($somme);
}

print "1+2 = " . somme (1, 2) . "\n";
print "1+2+3+4 = " . somme (1, 2, 3, 4) . "\n";

[yann@ulysse 14]$
```

Ecrivez et testez ce petit programme.

```
[yann@ulysse 14]$ cat exemple_sub_4.pl
#! /usr/bin/perl -w

sub produit_vectoriel
{
        (@u[0..2], @v[0..2]) = @_;
        $w [0] = $u[1] * $v[2] - $u[2] * $v[1];
        $w [1] = $u[2] * $v[0] - $u[0] * $v[2];
        $w [2] = $u[0] * $v[1] - $u[1] * $v[0];
        return (@w);
}

sub affiche_vecteur
{
        ($x, $y, $z) = @_;
        return ("($x, $y, $z)");
}

@i = (1, 0, 0);
@j = (0, 1, 0);
@k = produit_vectoriel (@i, @j);

print affiche_vecteur (@i);
print " x ";
print affiche_vecteur (@j);
print " = ";
print affiche_vecteur (@k);
print "\n";
[yann@ulysse 14]$
```

7.7.3 Passage des arguments

Le passage des arguments en Perl se fait toujours par référence. C'est à dire que dans sa table `@_`, la fonction a accès à la véritable variable qui se trouve sur la ligne d'invocation de la fonction. Toute modification de la table `@_` aura des répercutions au niveau supérieur d'exécution.
Le programme suivant permet de tester ceci. Que remarquez vous :

```
[yann@ulysse 14]$ cat exemple_sub_5.pl
#! /usr/bin/perl -w

sub efface
{
        for ($i = 0; $i < @_; $i++) {
                $_[$i] = 0;
        }
}

$a = 4;
$b = 5;
print "a=$a b=$b\n";
efface ($a, $b);
print "a=$a b=$b\n";
```

```
efface (12);
[yann@ulysse 14]$
```

Ecrivez et testez ce petit programme. Que remarquez vous ?

```
[yann@ulysse 14]$ cat exemple_sub_6.pl
#! /usr/bin/perl -w

sub efface
{
        @args=@_;
        for ($i = 0; $i < @args; $i++) {
                $args[$i] = 0;
        }
}

$a = 4;
$b = 5;
print "a=$a b=$b\n";
efface ($a, $b);
print "a=$a b=$b\n";
efface (12);
[yann@ulysse 14]$
```

7.7.4 Portée des variables

7.7.4.1 Variables globales

Par défaut en Perl, les variables déclarées dans un bloc d'instructions sont globales. On pourra y accéder depuis n'importe quelle autre partie du script.
L'exemple suivant illustre ceci :

```
[yann@ulysse 14]$ cat exemple_variables_1.pl
#! /usr/bin/perl -w

$a = "précédent";

print "Avant : a=$a, b=$b\n";
fonction();
print "Après : a=$a, b=$b\n";

sub fonction
```

```
{
        $a="suivant";
        $b="nouveau";
}
[yann@ulysse 14]$
```

Executez ce script. Que remarquez vous ?

7.7.4.2 Variables locales

Deux mots clés, `my` et `local` permettent de définir des variables locales. La différence la plus évidente entre les deux est que `my` définit une variable locale qui n'est accessible que dans le bloc d'instructions auquel elle appartient. Alors qu'avec `local`, la variable sera également visible et modifiable dans les sous fonctions.
L'exemple suivant illustre ceci :

```
[yann@ulysse 14]$ cat exemple_variables_2.pl
#! /usr/bin/perl -w

fonction();

sub fonction
{
        my $a = "initiale";
        local $b = "initiale";

        print "fonction() : a=$a, b=$b\n";
        fonction_2();
        print "fonction() : a=$a, b=$b\n";
}

sub fonction_2
{
        $a="modifiée";
        $b = "modifiée";
        print "fonction_2() : a=$a, b=$b\n";
}

[yann@ulysse 14]$
```

Que remarquez vous ?

7.7.5 Référence symbolique de routines

Perl permet d'utiliser les références symboliques sur les noms de fonction. Il autorise aussi l'emploi de références physiques. La notation pour invoquer une routine dont le nom est stocké dans la variable `$nom_fonction` est :

```
&$nom_fonction(argument);
```

L'exemple suivant illustre nos propos :

```
[yann@ulysse 14]$ cat exemple_references.pl
#! /usr/bin/perl -w

my $nom_fonction="factorielle";

my $resultat=&$nom_fonction(5);

print "$resultat\n";

sub factorielle
{
        my ($val) = @_;
        return 1 if ($val <= 1);
        return $val * factorielle($val -1);
}

[yann@ulysse 14]$
```

Testez ce programme.

7.7.6 Prototypes

Afin de s'assurer qu'une fonction est bien invoquée avec les arguments corrects, il est possible de fournir un prototype dans sa déclaration. Chaque argument est représenté par un caractère qui indique son type.

Caractère	Type d'argument
$	scalaire
@	liste
%	hachage
\$	variable scalaire
\@	variable table
\%	variable table hachage
*	descripteur de fichier
&	sous programme anonyme

Cette page est laissée blanche intentionnellement

Chapitre 8

Unix Avancé : Gestion de processus avec `fork`

Cette partie du TP est issue de TPs existant (B. Dupouy et S. Gadret) disponible à l'adresse suivante : http://www.infres.enst.fr/~domas/BCI/Proc/TPproc.html

8.1 La compilation sous Unix

Créer un répertoire dans lequel vous travaillerez, par exemple `tpsysteme`. Vous y placerez les fichiers à compiler ainsi que les exécutables.
Par la suite, pour compiler les fichiers, utiliser la commande `gcc`, par exemple (premier exercice) : `gcc exo1.c -o exo1` ou `gcc -Wall exo1.c -o exo1`. Pour plus de détail taper la commande `man gcc`

8.2 Création de Processus : `fork`

8.2.1 Fonctions utilisées

On va utiliser les fonctions Unix suivantes :

- `fork()` Cette fonction va créer un processus. La valeur de retour n de cette fonction indique :
  - $n > 0$ On est dans le processus père
  - $n = 0$ On est dans le processus fils
  - $n = -1$ fork a échoué, on n'a pas pu créer de processus
- getpid() : Cette fonction retourne le numéro du processus courant,
- getppid() : Cette fonction retourne le numéro du processus père.

8.2.2 Exercice 1

Taper ou récupérer le programme `exo1.c`.
Fichier `exo1.c`

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void main (void)
{
  int valeur;
  valeur = fork();
  printf (" Valeur retournee par la fonction fork: %d\n", (int)valeur);
  printf ("Je suis le processus numero %d\n", (int)getpid());
}
```

Après compilation de `exo1.c`, on exécutera le programme exo1 plusieurs fois. Que se passe-t-il ? Pourquoi ?

Réponse :

Ajouter maintenant la ligne suivante derrière l'appel à `fork` :

```
if (valeur == 0) sleep (4);
```

Que se passe-t-il ? Pourquoi ?

Réponse :

8.2.3 Exercice 2

Taper ou récupérer le programme `fork-mul.c`.
Fichier `fork-mul.c`

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void main (void)
{

  int valeur, valeur1 ;
  printf (" print 1 - Je suis le processus pere num=%d \n",
      (int)getpid() );
  valeur = fork();
  printf (" print 2 - retour fork: %d - processus num= %d -num pere=%d \n",
      valeur, (int)getpid(), (int)getppid() );
  valeur1 = fork();
  printf (" print 3 - retour fork: %d - processus num= %d -num pere=%d \n",
      valeur1, (int)getpid(), (int)getppid() );
}
```

Compiler `fork-mul.c`, puis l'exécuter.

Après exécution et à l'aide du schéma suivant (Cf. figure 8.1), relever les numéros des processus et numéroter l'ordre d'exécution des instructions `printf` de façon à retrouver l'ordre d'exécution des processus.

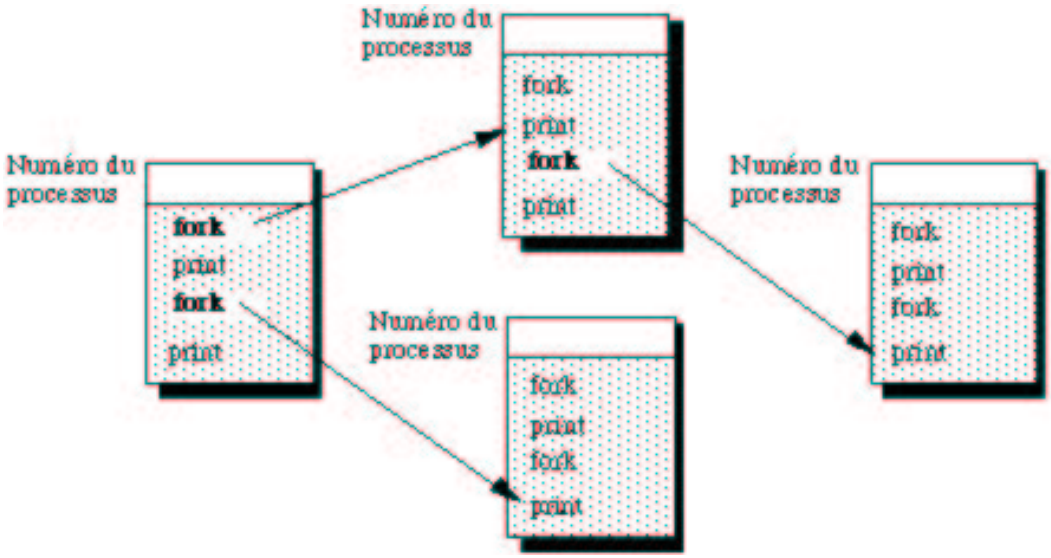


FIG. 8.1 – Hiérarchie des processus

Remarque si on relève un numéro de processus égal à 1, il s'agit du processus `init`, père de tous les processus. Init adopte les processus orphelins, c'est à dire qu'un processus dont le père s'est terminé devient fils de 1, et `getppid()` renvoie 1.

Réponse :

8.3 Père et fils exécutent des programmes différents

8.3.1 Introduction

Le moyen de créer un nouveau processus dans le système est l'appel de la fonction `fork`, qui duplique le processus appelant. De même, la seule manière d'exécuter un nouveau programme est d'appeler l'une des fonctions de la famille `exec()`.
L'appel à ces fonctions permet de remplacer l'espace mémoire du processus appelant par le code et les données de la nouvelle application. Ces fonctions ne reveinnent qu'en cas d'erreur, sinon le processus appelant est complètement remplacé.
On parle de l'appel système `exec()` sous forme générique, mais il n'existe aucune routine ayant ce nom. Il y a six variantes nommée `execl()`, `execle()`, `execlp()`, `execv()`, `execve()`, `execvp()`. Ces fonctions permettent de lancer une application. Les différences portent sur la manière de transmettre les arguments et l'environnement, et aussi sur la méthode pour accèder au programme à lancer. Il n'existe sous Linux qu'un seul appel-système dans cette famille de fonctions : `execve()`. Les autres fonctions sont implémentées à partir de cet appel système.
Les fonctions dont le suffixe commence par un `l` utilisent une liste d'arguments à transmettre de noms de variable, tandis que celles qui débutent par un `v` emploient un tableau à la manière du vecteur `argv[]`.

Les fonctions se terminant par un `e` transmettent l'environnement dans un tableau `envp[]` explicitement passé dans les arguments de la fonction, alors que les autres utilisent une variable `environ`.
Les fonctions se finissant par un `p` utilisent une variable d'environnement `PATH` pour rechercher le répertoire dans lequel se situe l'application à lancer, alors que les autres nécessitent un chemin d'accès complet. La variable `PATH` est déclarée dans l'environnement comme étant une liste de répertoire séparée par des deux-points.
Le prototype `execve()` est le suivant :

```
int execve (const char * appli, const char * argv [], const char * envp []);
```

La chaine `appli` doit contenir le chemin d'accès au programme à lancer à partir du répertoire de travail en cours ou à partir de la racine du système de fichiers s'il commence par un `/`.
Le tableau `argv []` contient des chaînes de caractères correspondant aux arguments que l'on trouve habituellement sur la ligne de commande.
La première chaine `arg[0]` doit contenir le nom de l'application à lancer (sans chemin d'accès).
Le troisième argument est un tableau de chaînes déclarant les variables d'environnement. On peut éventuellement utiliser la variable externe globale `environ` si on désire transmettre le même environnement au programme à lancer.
Les tableaux `argv[]` et `envp[]` doivent se terminer par des pointeurs `NULL`.
Récapitulons les caractéristiques des six fonctions de la famille `exec` :

- `execv()`
  - tableau `argv[]` pour les arguments ;
  - variable externe globale pour l'environnement ;
  - nom d'application avec chemin d'accès complet.
- `execve()`
  - tableau `argv[]` pour les arguments ;
  - tableau `envp[]` pour l'environnement ;
  - nom d'application avec chemin d'accès complet.
- `execvp()`
  - tableau `argv[]` pour les arguments ;
  - variable externe globale pour l'environnement ;
  - application recherchée suivant le contenu de la variable `PATH`.
- `execl()`
  - liste d'arguments `arg0`, `arg1` ... `NULL` ;
  - variable externe globale pour l'environnement ;
  - nom d'application avec chemin d'accès complet.
- `execle()`
  - liste d'arguments `arg0`, `arg1` ... `NULL` ;
  - tableau `envp[]` pour l'environnement ;
  - nom d'application avec chemin d'accès complet.
- `execlp()`
  - liste d'arguments `arg0`, `arg1` ... `NULL` ;
  - variable externe globale pour l'environnement ;
  - application recherchée suivant le contenu de la variable `PATH`.

8.3.2 Fonction utilisée

La fonction `exec` charge un fichier dans la zone de code du processus qui l'appelle, remplaçant ainsi le code courant par ce fichier. Une des formes de cette fonction est `execl` :

```
execl ("fic",arg0,arg1, (char *)0)
```

`fic` est le nom du fichier exécutable qui sera chargé dans la zone de code du processus.

8.3.3 Exemple avec `execvp()`

Ce petit exemple utilise simplement la commande `ls`.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <unistd.h>
#include <errno.h>

int main (void)
{
char * argv [] = { "ls", "-l", "-n", NULL };

execvp ("ls", argv);

fprintf (stderr, "Erreur %d\n", errno);
return (1);
}
```

La compilation est faite à l'aide de la ligne de commande suivante :

```
gcc -Wall -pedantic -g    exemple_execvp.c   -o exemple_execvp
```

Une liste non exhaustive des options de gcc est résumée dans la tableau suivant :

Options	Argument	But
-E		Arrêter la compilation après le passage du préprocesseur, avant le compilateur ;
-S		Arrêter la compilation après le passage du compilateur, avant l'assembleur ;
-c		Arrêter la compilation après l'assemblage, laissant les fichiers objets disponibles ;
-W	Avertissement	Valider les avertissements (*warnings*) décrits en arguments. Il en existe une multitude, mais l'option la plus couramment utilisée est `-Wall`, pour activer tous les avertissements ;
-pedantic		Le compilateur fournit des avertissements encore plus rigoureux qu'avec `-Wall`, principalement orientés sur la portabilité du code ;
-g		Inclure dans le code exécutable les informations nécessaires pour utiliser le débugueur. Cette option est généralement conservée jusqu'au basculement du produit en code de distribution ;
-O	0 à 3	Optimiser le code engendré. Le niveau d'optimisation est indiqué en argument (0=aucune). Il est déconseillé d'utiliser simultanément l'optimisation et le débogage.

Tester le programme, normalement, il vous affiche le contenu du répertoire courant.

```
yann@ulysse:~/projet_signal/prog_linux/04$ ./exemple_execvp
total 232
-rwxr-xr-x    1 1000     1000        22513 ao  6 15:50 exemple_execlp
-rw-r--r--    1 1000     1000          394 mar 22  2000 exemple_execlp.c
-rwxr-xr-x    1 1000     1000        23750 ao  6 15:50 exemple_execv
-rw-r--r--    1 1000     1000          942 mar 22  2000 exemple_execv.c
-rwxr-xr-x    1 1000     1000        22800 ao  6 15:50 exemple_execve
-rw-r--r--    1 1000     1000          351 mar 22  2000 exemple_execve.c
-rwxr-xr-x    1 1000     1000        22456 ao  6 15:55 exemple_execvp
-rw-r--r--    1 1000     1000          231 ao  6 15:54 exemple_execvp.c
-rwxr-xr-x    1 1000     1000        23598 ao  6 15:50 exemple_popen_1
-rw-r--r--    1 1000     1000          602 mar 22  2000 exemple_popen_1.c
-rwxr-xr-x    1 1000     1000        23469 ao  6 15:50 exemple_popen_2
-rw-r--r--    1 1000     1000          607 mar 22  2000 exemple_popen_2.c
-rwxr-xr-x    1 1000     1000        23626 ao  6 15:50 exemple_popen_3
-rw-r--r--    1 1000     1000         1497 mar 22  2000 exemple_popen_3.c
-rwxr-xr-x    1 1000     1000         1123 mar 22  2000 exemple_popen_3.tk
```

```
-rwxr-xr-x    1 1000     1000        17243 ao  6 15:50 exemple_system
-rw-r--r--    1 1000     1000           95 mar 22  2000 exemple_system.c
-rwxr-xr-x    1 1000     1000           52 mar 22  2000 ls
-rw-r--r--    1 1000     1000          295 mar 22  2000 Makefile
yann@ulysse:~/projet_signal/prog_linux/04$
```

Maintenant, nous allons changer le `PATH`.

```
yann@ulysse:~/projet_signal/prog_linux/04$ whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.gz
yann@ulysse:~/projet_signal/prog_linux/04$ export sauvpath=$PATH
yann@ulysse:~/projet_signal/prog_linux/04$ echo $sauvpath
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games:/usr/bin
yann@ulysse:~/projet_signal/prog_linux/04$ export PATH=/usr/bin
yann@ulysse:~/projet_signal/prog_linux/04$ ./exemple_execvp
Erreur 2
yann@ulysse:~/projet_signal/prog_linux/04$
```

Le programme ne semble plus marcher. Pourquoi : Réponse :

8.3.4 Fin d'un programme

Un processus peut se terminer normalement ou anormalement. Dans le premier cas, l'application est abandonnée à la demande de l'utilisateur, ou la tâche à accomplir est terminée. Dans le second cas, un dysfonctionnement est découvert, qui est si sérieux qu'il ne permet pas au programme de continuer son travail.

8.3.4.1 Terminaison normale d'un processus

Un programme peur se terminer de plusieurs manières. La plus simple est de revenir de la fonction `main` en renvoyant un compte rendu d'exécution sous forme de valeur entière. Cette valeur est lue par le processus père qui peut en tirer les conséquences adéquates. Par convention, un programme qui réussit à effectuer son travail renvoie une valeur nulle, tandis que les cas d'échecs sont indiqués par des codes de retour non nuls.
Si seuls, la réussite ou l'échec du programme importent, il est possible d'employer les constantes symboliques `EXIT_SUCCESS` ou `EXIT_FAILURE` définies dans `<stdlib.h>`.
Une autre manière de terminer un programme normalement est d'utiliser la fonction `exit()`.

```
void exit(int code);
```

On lui transmet en argument le code de retour pour le processus père. L'effet est strictement égal à celui d'un retour depuis la fonction `main`, à la différence que `exit()` peut être invoquée depuis n'importe quelle partie de programme.
Soit le programme `exemple_exit_1.c` suivant :

```
#include <stdlib.h>
void    sortie (void);

int main (void)
{
        sortie ();
}

void sortie (void)
{
        exit (EXIT_FAILURE);
}
```

Compilez le à l'aide de la commande :

```
gcc -Wall exemple_exit_1.c -o exemple_exit_1
```

Que remarquez vous ?
Réponse :

Comment faire pour enlever ce message.

Réponse :

8.3.4.2 Terminaison anormale d'un processus

Un programme peut aussi se terminer de manière anormale. Ceci est le cas lorqu'un processus exécute une instruction illégale, ou qu'il essaye d'accéder au contenu d'un pointeur mal initialisé. Ces actions déclenche un signal qui, par défaut, arrête le processus en créant un fichier d'image mémoire `core`.
Une manière propre d'interrompre anormalement un programme est d'invoquer la fonction `abort`.

```
void abort(void);
```

Celle-ci envoie immédiatement au processus le signal `SIGABRT`, en lé débloquant s'il le faut.
Le problème de la fonction `abort` ou des arrêts dus à des signaux est qu'il est difficile de déterminer ensuite à quel endroit du programme le dysfonctionnement a eu lieu. Il est toujours possible d'autopsier le fichier `core`, mais ceci est parfois ardu.
Une autre manière de détecter automatiquement les bogues est d'utiliser systématiquement la fonction `assert()` dans les parties critiques du programme. Il s'agit d'une macro définie dans `<assert.h>`, et qui évalue l'expression qu'on lui transmet en argument. Si l'expression est vraie, elle ne fait rien. Par contre, si elle est fausse, `assert()` arrête le programme après avoir écrit un message sur la sortie d'erreur standard, indiquant le fichier source concerné, la ligne de code et le texte de l'assertion ayant échoué. Il est alors très facile de se reporter au point décrit pour rechercher le bogue.
La macro `assert()` agit en surveilant perpétuellement que les conditions prévues pour l'éxecution du code soient respectées.
Soit le fichier source `exemple_assert.c` :

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

void    fonction_reussissant (int i);
void    fonction_echouant    (int i);

int main (void)
{
        fonction_reussissant (5);
        fonction_echouant (5);
```

```
        return (EXIT_SUCCESS);
}

void fonction_reussissant (int i)
{
        /* Cette fonction nécessite que i soit positif */
        assert (i >= 0);
        fprintf (stdout, "Ok, i est positif\n");
}

void fonction_echouant (int i)
{
        /* Cette fonction nécessite que i soit négatif */
        assert (i <= 0);
        fprintf (stdout, "Ok, i est négatif\n");
}
```

Compilez le programme à l'aide de la commande :

```
gcc -Wall -g exemple_assert.c -o exemple_assert
```

Lancez le programme. Que se passe t il ?
Réponse :

8.3.4.3 Attendre la fin d'un processus fils

L'une des notions fondamentales dans la conception des systèmes UNIX est la mise à disposition de l'utilisateur d'un très grand nombre de petits utilitaires très spécialisés et très configurables grâce à eds options de la ligne de commande. Ces petits utilitaires peuvent être associés, par des redirections d'entrées sorties, en commandes plus complexes et regroupés dans des fichiers scripts simples à écrire et à déboguer.
Il est primordial dasn ces scripts de pouvoir déterminer si une commande à réussi à effectuer son travail correctement ou non. Une grande important doit donc être portée à la lecture du code de retour d'un processus. Cette importance est telle qu'un processus qui se termine passe automatiquement par un état spécial, zombie,en attendant que le processus père ait lu son code de retour. Si le processus père ne lit pas le code de retour de son fils, ce dernier peut rester indéfiniment à l'état de zombie.
L'exemple suivant illustre le phénomène décrit ci-dessus. Siot le fichier source `exemple_zombie_1.c` :

```
        #include <stdio.h>
        #include <stdlib.h>
        #include <unistd.h>

int main (void)
{
        pid_t pid;
        char  commande [128];

        if ((pid = fork()) < 0)
{
```

```
                fprintf (stderr, "echec fork()\n");
                exit (1);
        }

        if (pid == 0)
{
                /* processus fils */
                sleep (2);
                fprintf (stdout, "Le processus fils %u se termine\n", getpid());
                exit (0);
        }
else
{
                /* processus père */
                sprintf (commande, "ps %u", pid);
                system (commande);
                sleep (1);
                system (commande);
                sleep (1);
                system (commande);
                sleep (1);
                system (commande);
                sleep (1);
                system (commande);
                sleep (1);
                system (commande);
        }
        return (0);
}
```

Compilez et lancez le programme. Que se passe t il ?
Une fois le programme terminé, lancez la commmande `ps`. Le processus fils est il toujours présent ?

Réponse :

L'exemple suivant décrit un autre phénomène. Le fichier source `exemple_zombie_2.c` est le suivant :

```
#include <stdio.h>
```

```
        #include <stdlib.h>
        #include <unistd.h>

        int
main (void)
{
        pid_t pid;

        fprintf (stdout, "Père : mon PID est %u\n", getpid());

        if ((pid = fork()) < 0) {
                fprintf (stderr, "echec fork()\n");
                exit (1);
        }


        if (pid != 0) {
                /* processus père */
                sleep (2);
                fprintf (stdout, "Père : je me termine\n");
                exit (0);
        } else {
                /* processus fils */

                fprintf (stdout, "Fils : mon père est %u\n", getppid ());
                sleep (1);
                fprintf (stdout, "Fils : mon père est %u\n", getppid ());
                sleep (1);
                fprintf (stdout, "Fils : mon père est %u\n", getppid ());
                sleep (1);
                fprintf (stdout, "Fils : mon père est %u\n", getppid ());
                sleep (1);
                fprintf (stdout, "Fils : mon père est %u\n", getppid ());
        }
        return (0);
}
```

Compilez et lancez le programme. Que se passe t il ?

Réponse :

Pour lire le code de retour d'un processus fils, il existe quatre fonctions : `wait()`, `waitpid()`, `wait3()` et `wait4()`. Les trois premières sont des fonctions de bibliothèque implémentées en invoquant `wait4()` qui est le seul véritable appel système.
Nous n'étudierons que la fonction `wait()`.

La fonction `wait()` est déclarée dans `<sys/wait.h>`, ainsi :

```
pid_t wait(int * status);
```

Lorsqu'on l'invoque, elle bloque le processus appelant jusqu'à ce qu'un de ses fils se termine. Elle renvoie alors le PID du fils terminé. Si le pointeur `status` est non NULL, il est renseigné avec une valeur informant sur les circonstances de la mort du fils. Si un processus fils était déjà en attente à l'état de zombie, `wait()` revient immédiatement. Si les corconstances de la fin du processus ne nous intéressent pas, il est possible de fournir un argument NULL. La manière dont sont organisées les informations au sein de l'entier `status` est opaque, et il faut utiliser les macros suivantes pour analyser les circonstances de la fin d'un processus fils :

- `WIFEXITED(status)` est vraie si le processus s'est terminé de son propre chef en invoquant `exit()` ou en revenant de `main()`. On peut alors obtenir le code de retour du fils en invoquant `WEXITSTATUS(status)` ;
- `WIFSIGNALED(status)` indique que le fils s'est terminé à cause d'un signal, y compris le signal `SIGABRT`, envoyé lorsqu'il appelle `abort()`. Le numéro de signal ayant tué le processu fils est disponible en utilisant la macro `WTERMSIG(status)`. À ce moment,la macro `WCOREDUMP(status)` signal si une image mémoire `core` a été créée ;
- `WIFSTOPPED(status)` indique si le fils a ét stopppé temporairement . Le numéro de signal ayant stoppé le processus fils est accessible en utilisant `WSTOPSIG(status)`.

Dans l'exemple qui suit, le processus père va se dédoubler en une série de fils qui se termineront de manière variées. Le processus père restera en bouble sur `wait()`, jusqu'à ce qu'il ne reste plus de fils. Voici le fichier source :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>


void    affichage_type_de_terminaison (pid_t pid, int status);
int     processus_fils(int numero);

int main (void)
{
pid_t   pid;
int     status;
int     numero_fils;

        for (numero_fils = 0; numero_fils < 4; numero_fils ++) {

                switch (fork ()) {
                        case -1 :
                                fprintf (stderr, "Erreur dans fork()\n");
                                exit (1);
                        case 0 :
                                fprintf (stdout, "Fils %d : PID = %u\n",
                                        numero_fils, getpid ());
                                return (processus_fils (numero_fils));
                        default :
                                /* processus père */
                                break;
                }
        }

        /* Ici il n'y a plus que le processus père */

        while ((pid = wait (& status)) > 0)
                affichage_type_de_terminaison (pid, status);
```

```
        return (0);
}

void affichage_type_de_terminaison (pid_t pid, int status)
{
        fprintf (stdout, "Le processus %u ", pid);
        if (WIFEXITED (status))
{
                fprintf (stdout, "s'est terminé normalement avec le code %d\n",
                                 WEXITSTATUS (status));
        }
else if (WIFSIGNALED (status))
{
                fprintf (stdout, "s'est terminé à cause du signal %d (%s)\n",
                                WTERMSIG (status),
                                sys_siglist [WTERMSIG (status)]);
                if (WCOREDUMP (status)) {
                        fprintf (stdout, "Fichier image core créé\n");
                }
        }
else if (WIFSTOPPED (status))
{
                fprintf (stdout, "s'est arrêté à cause du signal %d (%s)\n",
                                WSTOPSIG (status),
                                sys_siglist [WSTOPSIG (status)]);
        }
}

int processus_fils (int numero)
{
        switch (numero)
{

                case 0 :
                        return (1);
                case 1 :
                        exit (2);
                case 2 :
                        abort ();
                case 3 :
                        raise (SIGUSR1);
        }
        return (numero);
}
```

Compilez et lancez le programme. Que se passe t il ?

Réponse :

8.4 Synchronisation de processus père et fils (mécanisme wait/exit)

8.4.1 Fonctions utilisées

– `exit(i)` termine un processus, `i` est un octet (donc valeurs possibles : 0 à 255) renvoyé dans une variable du type `int` au processus père.
– `wait(&Etat)` met le processus en attente de la fin de l'un de ses processus fils .
La valeur de retour de `wait` est le numéro du processus fils venant de se terminer. Lorsqu'il n'y a plus (ou pas) de processus fils à attendre, la fonction wait renvoie -1. Chaque fois qu'un fils se termine le processus père sort de `wait`, et il peut consulter `Etat` pour obtenir des informations sur le fils qui vient de se terminer. `Etat` est un pointeur sur un mot de deux octets. L'octet de poids fort contient la valeur renvoyée par le fils (`i` de la fonction `exit(i)`), et l'octet de poids faible contient 0.
En cas de terminaison anormale du processus fils, l'octet de poids faible contient la valeur du signal reçu par le fils. Cette valeur est augmentée de 80 en hexadécimal (128 décimal), si ce signal a entrainé la sauvegarde de l'image mémoire du processus dans un fichier `core`. Contenu du mot `Etat` à la sortie de `wait` (Cf. figure 8.2) :

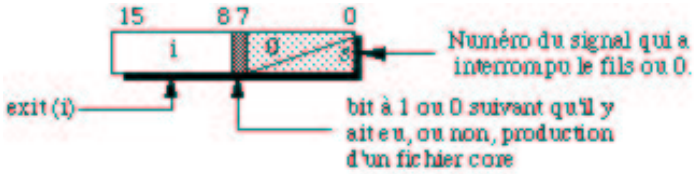


FIG. 8.2 – Retour de la fonction `wait`

8.4.2 Mécanismes `wait/exit`

Taper ou récupérer le fichier `fork-sync.c`. Compiler et exécuter le.
Fichier `fork-sync.c`

```
#include <stdio.h> #include <sys/types.h> #include <unistd.h>
#include <sys/wait.h>

void main (void) {
  int valeur, ret_fils,etat ;
  printf ("Je suis le processus pere num=%d \n", (int)getpid());
  valeur=fork();
  switch (valeur)
    {
    case 0 :
      printf
```

```
    ("\t\t\t\t********\n\t\t\t\t* FILS *    \n\t\t\t\t********\n");
      printf ("\t\t\t\tProc fils num= %d - \n\t\t\t\tPere num= %d \n",
          (int) getpid(),(int) getppid() );
      printf("\t\t\t\tJe vais dormir 30 secondes ...\n");
      sleep (30);
      printf
    ("\t\t\t\tJe me reveille ,
          \n\t\t\t\tJe termine mon execution par un EXIT(7)\n");
      exit (7);
    case -1:
      printf ("Le fork a echoue");
      exit(2);
    default:
      printf("********\n* PERE *\n********\n");
      printf ("Proc pere num= %d -\n Fils num= %d \n",
          (int) getpid(),valeur );
      printf ("J'attends la fin de mon fils: \n");
      ret_fils = wait (&etat);
      printf
    ("Mon fils de num=%d est termine,\nSon etat etait :%0x\n",
     ret_fils,etat);
    }
}
```

Que se passe-t-il ? Pourquoi ?

Réponse :

8.4.3 Fonctionnement de exec

Taper ou récupérer le fichier `fexec.c`. Compiler puis exécuter le. On pourra y faire exécuter par `exec` un fichier très simple du type :

```
void main (void)
{
   printf(" Coucou, ici %d ! \n", getpid() );
   sleep (4);
}
```

Fichier `fexec.c`

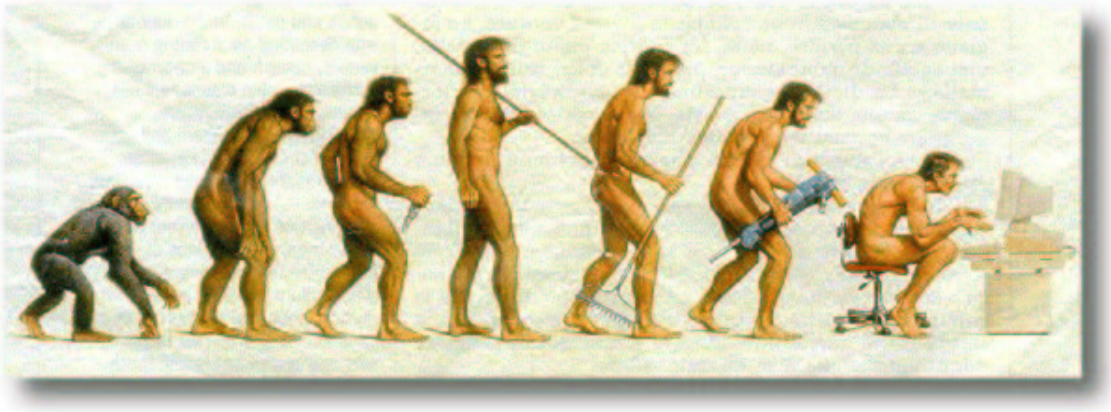
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

void main (int argc, char *argv[])
{
  int Pid;
  int Fils,Etat;
  /*
    ------------------------------------------------------------
```

```
    On peut executer ce programme en lui passant diverses
    commandes en argument, par exemple, si l'executable est fexec :
    fexec /usr/bin/ps
    ------------------------------------------------------
  */

  if (argc != 2)
    {printf(" Utilisation : %s fic. a executer ! \n", argv[0]);
    exit(1);
    }

  printf (" Je suis le processus %d je vais faire fork\n",(int) getpid());
  Pid=fork();
  switch (Pid)
    {
    case  0 :
      printf (" Coucou ! je suis le fils %d\n",(int) getpid());
      printf (" %d : Code  remplace par %s\n",(int) getpid(), argv[1]);
      execl(argv[1],(char *)0);
      printf (" %d : Erreur lors du exec \n", (int) getpid());
      exit (2);

    case   -1 :
      printf (" Le fork n'a pas reussi ");
      exit (3) ;

    default :
      /* le pere attend la fin du fils */
      printf (" Pere numero %d attend\n ",(int) getpid());
      Fils=wait (&Etat);
      printf ( " Le fils etait : %d ", Fils);
      printf (" ... son etat etait :%0x (hexa) \n",Etat);
      exit(0);
    }
}
```

Que se passe-t-il ? Pourquoi ?

Réponse :

Vous en avez maintenant terminé avec Unix...



Réflexion sur l'évolution humaine...

Bibliographie

[1] Axis & Agix. *Unix Utilisation : Guide de Formation*. EDITIONS LASER, 1995.

[2] CH. Blaess. *Langages de scripts sous Linux*. Eyrolles, 2002.

[3] CH. Blaess. *Programmation Système en C sous Linux*. Eyrolles, 2002.

[4] J.M. Champarnaud and G. Hansel. *Passeport pour UNIX et C*. Passeport pour l'informatique. Internationnal Thomson Publishing, 1995.

[5] P. Charman. *Unix et X-Window : Guide Pratique*. CÉPADUÈS ÉDITIONS, 1994.

[6] T. Poulain. Cours unix. 1994.

[7] Richard Stoeckel. *Filtres et Utilitaires Unix*. ARMAND COLIN, 1992.