Université de Metz

U.F.R. SciF.A.

Nom :_____ **Prénom :**_____

Groupe :_____

T.P. de Système d'Exploitation Unix

Maîtrise E.E.A.

Année Universitaire 2004/2005

Y. Morère

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.
- ☑ Comprendre et maîtriser la programmation de scripts (`bash`, `Perl` pour automatiser des tâches
- ☑ Comprendre et maîtriser la programmation multithreading.

Évaluation et notation :

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

Table des matières

Table des figures

Cette page est laissée blanche intentionnellement

Liste des tableaux

Cette page est laissée blanche intentionnellement

Avant de commencer un peu de lecture

Procédure de Login

Suivant la salle où vous vous trouvez (ici c'est la salle Linus, en hommage au créateur du noyau Linux), le type de distribution Linux installée sur la machine, de petites différences dans le mode de connexion peuvent survenir.
Pour se connecter à l'une des machines, il faut bien entendu, avoir un accès à celle-ci c-à-d avoir un compte.
La plupart du temps, pour se connecter, il suffit de donner son nom de login et son mot de passe au logiciel de connexion (la plupart du temps `xdm`, `gdm` ou encore `kdm`). Vous obtenez une nouvelle fenêtre ressemblant à la figure 1.

FIG. 1 – Connexion à une machine en mode graphique (ici Ulysse avec `gdm`)

Login : saisissez ici votre **identifiant** ou **login** et `enter`
Password : saisissez ici votre **mot de passe** (Attention les lettres frappées s'affichent sous la forme d'étoiles) et `enter`.
Vous vous trouvez alors dans l'environnement LINUX, et vous avez une fenêtre du type de la figure 2.

FIG. 2 – Connexion réussie vous pouvez travailler

Toute la puissance d'un OS multitâches multiutilisateurs est a vous. Mais saurez-vous en tirer partie ?

Remarque sur le mot de passe

✗ Il faut impérativement changer le mot de passe après la première connexion.

- Vous seul connaissez le mot de passe.
- Si vous l'oubliez, la seule solution est de contacter l'administrateur réseau pour l'effacement de l'ancien mot de passe.
- Le mot de passe est composé d'au moins 6 caractères et au maximum 8 dont 2 au moins ne sont pas des lettres.
- Conseil : mélanger les lettres, chiffres, majuscules et minuscules.

Pour changer de mot de passe

- *yppasswd* ou *passwd* et `enter`
- La machine vous demande l'**ancien** mot de passe et `enter`
- Elle vous demande le **nouveau** et `enter`
- Puis elle redemande le **nouveau** pour confirmation et `enter`

Si vous n'avez pas de message d'erreur, le mot de passe est alors changé.

Terminer votre session

Pour terminer votre session LINUX (se *deloger*), il vous suffit de terminer la session graphique à l'aide du menu approprié. Je ne donnerai aucun détail, car ceci est très variant d'un gestionnaire de fenêtre à un autre.

Forme et validation d'une commande

Une ligne de commande possède la forme générale suivante :

nom-de-commande [options] arguments

où l'*option* a la forme *-lettre*.

Exemple : Dans la commande
`ls -s -i toto titils` est le nom de la commande, `-s -i` sont les options, `toto` et `titi` sont les arguments.
Plusieurs options peuvent être regroupées derrière le signe `-`.
`ls -si toto titi`
Une ligne de commande n'est reçue et exécutée par le système qu'après *validation* par `enter`.

Commande importante : `man`

La syntaxe est la suivante :
`man` *commande*
ou
`man -k` *commande*
Cette commande affiche la page de manuel correspondante à la *commande*. On y trouve tous les détails concernant son utilité, sa syntaxe et ses options.

Exemple : réponse du système à la commande `man ls` sur la machine `OSF1 flore V4.0 1091 alpha`

```
ls(1)

NAME

  ls - Lists and generates statistics for files

SYNOPSIS

  ls [-aAbcCdfFgilLmnopqrRstux1] [file...|directory...]

STANDARDS

  Interfaces documented on this reference page conform to industry standards
  as follows:
```

```
ls:  XPG4, XPG4-UNIX

Refer to the standards(5) reference page for more information about indus-
try standards and associated tags.
```

Pour toutes questions au sujet de la commande `man`, taper :

```
man man
```

Cette page est laissée blanche intentionnellement

Chapitre 1

Travaillons un peu

1.1 Introduction

1.1.1 Reconnaître votre `shell`

`echo $SHELL`
Réponse :

1.1.2 Reconnaître votre système d'exploitation

`uname -a`
Réponse :

1.2 Commandes Générales

1.2.1 Changer votre mot de passe

`passwd` ou `yppasswd`
Réponse :

Observations :

1.2.2 Afficher la ligne (terminal) sur laquelle vous êtes connectés

`tty`
Réponse :

1.2.3 Afficher les paramètres de votre terminal

`stty`
Réponse :

Observations :

1.2.4 Reconfigurer la touche `erase`

`stty erase ^H`
Réponse :

1.2.5 Qui êtes-vous ?

`whoami`
Réponse :

`who am i`
Réponse :

Observations :

1.2.6 Qui est connecté sur la station ?

`who`
Réponse :

`w`
Réponse :

`users`
Réponse :

Observations :

1.2.7 Afficher du texte et les valeurs des variables système

`echo "Bonjour"`
Réponse :

`echo $DISPLAY`
Réponse :

1.2.8 Afficher les variables de votre système

`env`
Réponse :

Observations :

1.2.9 Création d'un fichier simple

Saisissez les lignes suivantes
```
cat > fic.txt
mon premier fichier <enter>
bbbbbbbbbbbbbbbbbbb <enter>
ccccccccccccccccccc <enter>
aaaaaaaaaaaaaaaaaaa <enter>
ctrl-d
```
Réponse :

1.2.10 Visualisation de votre fichier

```
cat fic.txt
```
Réponse :

```
more fic.txt
```
Réponse :

`less fic.txt`
Réponse :

Observations :

1.2.11 Compter le nombre de lignes, de mots, de caractères du fichier

`wc fic.txt`
Réponse :

1.2.12 Recherche d'une chaîne dans un fichier

`grep aa fic.txt`
Réponse :

Observations :

1.2.13 Trier le contenu du ou des fichiers passés en argument

`sort fic.txt`
Réponse :

Cette page est laissée blanche intentionnellement

Chapitre 2

Passons la vitesse supérieure

2.1 Système de fichiers

2.1.1 Les caractères spéciaux

Il existe plusieurs caractères qui sont traités d'une manière différente pas l'interpréteur de commande. On les appelle des **caractères spéciaux** ou **métacaractères**. Ils comprennent :
- des caractères d'abréviations ;
- des caractères spéciaux liés au lancement d'une commande.

2.1.1.1 Les caractères d'abréviations

`*` remplace toute chaîne de caractères permettant de compléter un nom de fichier sauf une chaîne commençant par le caractère "`.`" lequel doit être explicitement écrit.

`?` peut représenter un caractère quelconque (à l'exception de "`.`").

`[`***liste de caractères***`]` désigne un caractère quelconque écrit entre crochet. Par exemple `foo.[cp]` désigne la liste des 2 fichiers `foo.c` et `foo.p`.

2.1.1.2 Les caractères spéciaux

`;` sépare les instructions successives d'une ligne de commande.

`()` servent à regrouper les commandes.

`>` redirection de la sortie standard.

`>>` redirection de la sortie standard sans écrasement.

`2>` redirection de l'erreur standard.

`2>>` redirection de l'erreur standard sans écrasement.

`<` redirection de l'entrée standard.

`&` lancement d'un processus en arrière plan (tâche de fond).

`|` communication par tube entre deux processus.

Exemple :
- La commande `ls *.txt` permet de lister tous les fichiers possédant l'extension `txt`.
- La commande `ls t?t?.txt` permet de lister tous les fichiers `titi.txt`, `toto.txt`, `tctc.txt` ...
- La commande `ls t[ao]t[ao].txt` permet de lister tous les fichiers possédant l'extension `tata.txt`, `tota.txt`, `toto.txt`, `tato.txt`.

2.1.2 Quel est le répertoire courant

`pwd`
Réponse :

2.1.3 Lister les fichiers dans le répertoire courant

`ls`
Réponse :

`ls -l`
Réponse :

`ls -al`
Réponse :

Observations :

2.1.4 Copier un fichier

`cp fic.txt fic_copie.txt`
Réponse :

Vérification :

2.1.5 Renommer un fichier

`mv fic_copie.txt nouveau_nom.txt`
Réponse :

Vérification :

2.1.6 Effacer un fichier

`rm nouveau_nom.txt`
Réponse :

Vérification :
Recréer un fichier `nouveau_nom.txt` et effacer le par `rm -i nouveau_nom.txt`

Réponse :

Vérification :

Observations :

2.1.7 Créer un lien symbolique sur un fichier

`ln -s fic.txt fic_lien.txt`
Réponse :

Vérification :

2.1.8 Créer un lien sur un fichier

`ln fic.txt fic_lien.txt`
Réponse :

Vérification :

Observations par rapport au lien symbolique :

2.1.9 Visualiser le fichier `fic_lien.txt`

Réponse :

Effacer le fichier original `fic.txt`, Tester à nouveau les liens :
Observations entre le lien symbolique et le lien simple :

2.1.10 Créer un répertoire

`mkdir mon_repertoire`
Réponse :

Vérification :

2.1.11 Changer de répertoire

`cd mon_repertoire`
Réponse :

Vérification :

2.1.12 Revenir au répertoire précédent

`cd ..`
Réponse :

Vérification :

2.1.13 Effacer un répertoire

`rmdir mon_repertoire`
Réponse :

Vérification :

2.1.14 Effacer un répertoire : le retour

Créer un répertoire `parent`, se déplacer dans celui-ci, créer un répertoire `enfant`
Revenir dans votre répertoire `home` et lancer :
`rm -r parent`
Réponse :

Vérification :

Observations :

2.1.15 Comprendre les droits d'accès

La commande `ls -al` affiche le résultat suivant :

```
drwxr-xr-x 5 ymorere prof 1024 Sep 21 14 :57 public_html
```

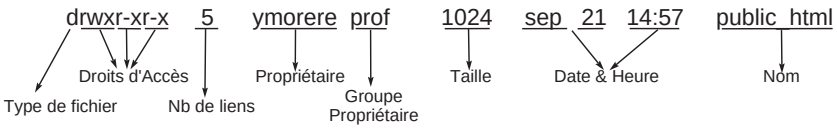


FIG. 2.1 – Signification des champs du résultat de la commande `ls -al`

Les droits d'accès sont les ensembles d'autorisations, qui déterminent *qui* peut avoir accès aux fichiers *en vue de quelle utilisation*.

droits d'accès	**utilisateur**s	**commande**s
lecture = **r**	propriétaire = **u**	ajouter = **+**
écriture = **w**	groupe = **g**	enlever = **-**
exécutable = **x**	autre = **o**	initialiser = **=**
pas d'accès = **-**	tous = **a**	

TAB. 2.1 – Tableau récapitulatif des droits d'accès

2.1.16 Modification des droits d'accès

Commande `chmod` *utilisateurs+/-droit fichier/répertoire*

Changer les droits d'accès de votre fichier `fic.txt` : vous devez être le seul à pouvoir lire et modifier votre fichier.

Réponse :

Vérification :

Observations :

Créer un répertoire et visualiser ses droits d'accès. Modifier ses droits d'accès afin que tout le monde puisse accéder à celui-ci mais ne puisse pas y faire de modification.

Réponse :

Vérification :

Observations :

2.1.17 Donner le type du fichier

Commande `file` *`nom_du_fichier`*
Exécuter la commande avec un fichier
Réponse :

Exécuter la commande avec un répertoire
Réponse :

Observations :

2.1.18 Trouver un fichier dans l'arborescence

Commande `find / -name "talk" -print`
recherche le fichier `talk` à partir du répertoire racine `/`
Rechercher les fichiers commençant par `x` dans le répertoire `/usr/bin`
Réponse :

2.1.19 Afficher les caractéristiques des disques

Commande `df`
Réponse :

Observations :

Chapitre 3

On a toujours besoin d'un éditeur de texte

3.1 Les éditeurs de texte

Un éditeur de texte est un outil pour écrire un fichier texte pur c'est à dire sans mise en forme (il ne contient que des caractères de la table ASCII) (`edit` sous DOS, `turbo` pour la programmation TurboPascal), alors qu'un traitement de texte est un outil pour écrire et mettre en forme du texte (*WordPerfect*, *MS Word*).

3.1.1 vi

`vi` est un éditeur vidéo, c'est à dire pleine page et interactif. Il est possible d'insérer du texte à tout endroit dans le fichier en édition. (Cf. figure 3.1)
L'appel de `vi` s'effectue de la manière suivante :

- `vi`
- `vi` *fichier*
- `vi` *+n fichier* pour se placer directement à la *n*-ième ligne du *fichier*
- `vi` *+/motif fichier* pour se placer directement à la première occurrence du *motif* dans le *fichier*

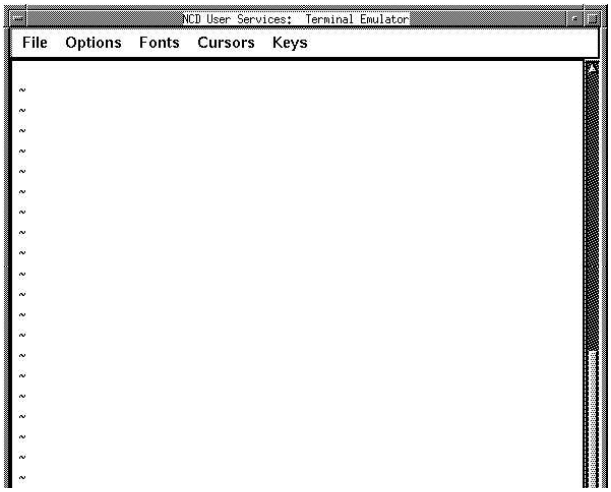


FIG. 3.1 – Fenêtre d'édition pleine page vi

Il dispose de 3 modes de travail :

Mode commande permet les déplacements, les recherches, les destructions, etc... C'est le mode par défaut.

Mode exécution permet l'utilisation de toutes les commandes de `ed` (editeur en ligne UNIX). Mode utilisé pour les commandes globales et les commandes gérant les fichiers.

Mode saisie permet la saisie de texte.

3.1.1.1 Mode saisie

Pour passer en mode saisie (depuis le mode commande) il faut taper une des commandes d'édition :
- `a` pour "append" (ajout) ajoute du texte après le curseur.
- `A` pour "append" (ajout) ajout du texte en fin de ligne.
- `i` pour "insert" (insertion) insère du texte devant le curseur.
- `I` pour "insert" (insertion) insère du texte en début de ligne.
- `o` pour "open" (ouvrir) ouvre une ligne après le curseur.
- `O` pour "open" (ouvrir) ouvre une ligne en fin de ligne.

3.1.1.2 Mode commande

Pour repasser en mode commande il faut appuyer sur `ESC`.

3.1.1.3 Déplacement du curseur

Déplacement du curseur en mode commande :
- `x`,`j`,`k` et `l` déplacent le curseur dans les 4 directions (O, S, N et E). Ces touches peuvent être remplacées par les flèches du clavier.
- `w` place le curseur au début du mot suivant
- `b` place le curseur au début du mot précédent
- `e` place le curseur à la fin du mot courant
- `0` place le curseur en début de ligne
- `$` place le curseur en fin de ligne
- `enter` place le curseur au début de la ligne suivante
- `^F` avance le curseur d'une page
- `^B` recule le curseur d'une page
- `G` place le curseur en fin de fichier
- *n*`G` place le curseur à la *n*-ième ligne du fichier
- `/`*motif*`/` place le curseur à la prochaine occurrence du *motif*
- `m`*c* définit une *marque* : elle associe la position du curseur au caractère *c*. On peut alors retourner d'un autre endroit du fichier à cette position par la commande `` `c `` ou au début de la ligne par `'c`

3.1.1.4 Commandes d'effacement et remise de texte

- `x` efface le caractère sous le curseur
- `dw` efface un mot
- `db` efface le mot précédent
- `D` efface la fin de la ligne
- `dd` efface la ligne du curseur
- `r`*c* remplace le caractère courant par *c*
- `~` remplace une minuscule par une majuscule et vice-versa
- `p` réinsère le texte dernièrement effacé

3.1.1.5 Insertion d'un fichier extérieur

- `:r` *fichier* insère après la ligne courante un fichier extérieur

3.1.1.6 Annulation de la dernière commande

- `u` annule la dernière commande effectuée

3.1.1.7 Recherche et remplacement

- `:/`*motif* place le curseur sur la prochaine occurrence de *motif*
- `:s/`*motif*`/`*chaîne*`/` remplace dans la ligne courante la première occurrence de *motif* par la *chaîne*
- `:s/`*motif*`/`*chaîne*`/g` remplace dans la ligne courante toute occurrence de *motif* par la *chaîne*
- `:1,10s/`*motif*`/`*chaîne*`/g` remplace toute occurrence de *motif* par la *chaîne* de la première à la dixième ligne
- `:.,$%s/`*motif*`/`*chaîne*`/g` remplace toute occurrence de *motif* par la *chaîne* depuis la ligne courante (désignée par `"."`) jusqu'à la dernière ligne du fichier (désignée par `"$"`)
- `:%s/`*motif*`/`*chaîne*`/g` remplace dans tout le fichier toute occurrence de *motif* par la *chaîne*

3.1.1.8 Déplacement de texte

- `m`*c* définit le début de la section à déplacer
- se déplacer à la fin de la section à déplacer et frapper `d'`*c*, la fraction de texte est alors supprimée et placée dans le tampon
- se déplacer à l'endroit voulu et frapper `p` qui insère le texte contenu dans le tampon.

3.1.1.9 Mode exécution

Le mode exécution est activé depuis le mode commande par la caractère ' :' suivi de la commande :
- `:w` *nom_fichier* sauvegarde le fichier *nom_fichier*
- `:w` sauvegarde du fichier
- `:wq` sauvegarde et quitte `vi`
- `:q!` quitte `vi` sans sauvegarde
- `:x` équivalent à `:wq`

3.1.2 emacs

Les principaux avantages d'`emacs` sont l'extensibilité, la personnalisation et l'auto-documentation. Il possède de nombreuses fonctionnalités autres que celles de l'édition.
On peut compiler un programme, lire du courrier électronique, lire les forums, récupérer un fichier par ftp... `emacs` signifie editor macros.
L'appel d'`emacs` s'effectue de la manière suivante :
- `emacs`
- `emacs` *fichier*
- `emacs` *+n fichier* pour se placer directement à la *n*-ième ligne du *fichier*

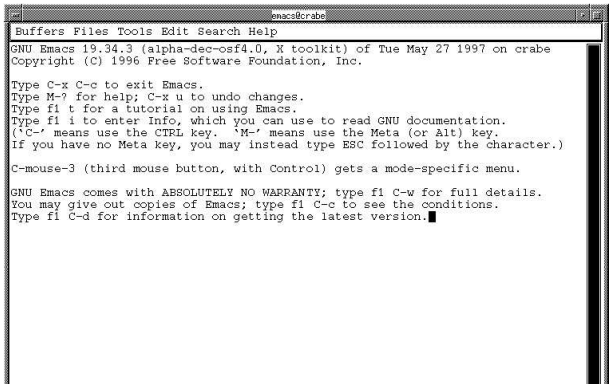


FIG. 3.2 – Fenêtre d'édition `emacs`

La plupart des commandes `emacs` font intervenir :
- la touche `<CTRL>` frappée en même temps qu'un autre caractère. Elle est habituellement notée `C-`. Par exemple `C-p` désigne la touche `<CTRL>` frappée en même temps que `p`.
- la touche `META`, existante sur la clavier, soit la touche `<ALT>` ou encore la touche `<ESC>` (dans ce cas elle doit être relâchée avant la touche qui suit). Elle est notée `M-`.

3.1.2.1 Les modes d'édition d'`emacs`

Selon que l'on travaille sur un fichier texte, un programme C, un fichier de données, les besoins d'édition sont différents. `emacs` propose donc plusieurs modes d'édition qui définissent un environnement de travail adapté au type de fichier.
Modes principaux :
- le mode **Fundamental**
- le mode **Text**
- le mode **Lisp**
- le mode **C**

Il existe des modes secondaires utilisés en conjonction avec un mode principal :

- le mode **Fill** : les lignes sont automatiquement coupées quand elles dépassent la marge droite.
- le mode **Abbrev** : l'expansion des abréviations est impossible.
- le mode **Ovwrt** : mode recouvrement.
- le mode **Narrow** : rend accessible qu'une partie du buffer.

3.1.2.2 Notion de buffer

C'est une partie de la mémoire d'`emacs`. A chaque ouverture de fichier un buffer portant le même nom est créé. Les modifications n'affectent pas tout de suite le fichier sur disque mais le buffer. Il faut sauver le fichier pour que les modifications soient effectuées sur le fichier d'origine.

3.1.2.3 Commande de déplacement du curseur

- `C-p` place le curseur sur la ligne précédente
- `C-n` place le curseur sur la ligne suivante
- `C-f` avance le curseur d'un caractère
- `M-f` avance le curseur à la fin du mot courant ou suivant
- `C-b` recule le curseur d'un caractère
- `M-b` recule le curseur au début du mot courant ou du précédent
- `C-a` place le curseur en début de ligne
- `C-e` place le curseur en fin de ligne
- `M-<` place le curseur en début de fichier
- `M->` place le curseur en fin de fichier
- `C-v` avance la fenêtre d'un écran
- `M-v` recule la fenêtre d'un écran

Toutes ces commandes peuvent être envoyées avec un argument numérique *n*. L'argument numérique est introduit par `C-u`.
La commande `C-u 8 C-n` avance le curseur de 8 lignes.

3.1.2.4 Insertion et suppression de texte

- `C-d` efface le caractère sur lequel se trouve le curseur
- `C-k` efface la fin de la ligne
- `C-y` réinsère le texte effacé
- `C-x u` annule l'effet de la dernière modification

3.1.2.5 Édition simultanée de plusieurs fichiers

- `C-x C-f` *fichier* crée un nouveau tampon et y place le *fichier*
- `C-x C-b` ouvre un nouveau tampon et une nouvelle fenêtre sur l'écran et y affiche la liste de tous les tampons ouverts (cette fenêtre peut être supprimée avec `C-x 1`)
- `C-x b` *tampon* place la curseur dans le *tampon*
- `C-x C-v` *fichier* place le *fichier* dans le tampon courant, son contenu actuel est éliminé.
- `C-x k` *tampon* supprime le *tampon* (par défaut c'est le tampon courant qui est supprimé)

3.1.2.6 Recherche et remplacement

- `C-s` *mot* recherche la première occurrence du *mot* dans la suite du texte. A chaque fois que l'utilisateur frappe une nouvelle lettre du *mot*, le curseur se place sur la prochaine occurrence de ce qui a été frappé. La répétition de `C-s` recherche l'occurrence du *mot* suivant.
- `C-r` *mot* a le même effet que `C-s`, mais la recherche se fait sur le texte qui précède.
- `M-%` *chaîne nouvelle_chaîne* remplace une *chaîne* de caractères par une *nouvelle_chaîne*. L'utilisateur doit valider chaque remplacement. S'il frappe `!` les validations sont omises

3.1.2.7 Insertion d'un fichier

- `C-x i` *fichier* insère le *fichier* à l'endroit du curseur

Pour n'insérer qu'une partie du fichier effectuer les commandes suivantes :
- `C-x C-f` *fichier* insère le *fichier* dans un nouveau tampon
- Placer le curseur au début de la région à insérer
- `C-@` ou `C-ESPACE` marque le début de la région
- Placer le curseur à la fin de la région à insérer
- `C-w` efface la région délimitée

- `C-x b` revient au tampon initial
- Se placer à l'endroit voulu de l'insertion
- `C-y` insère le contenu du tampon auxiliaire

3.1.2.8 Suppression de fenêtres

- `C-x 1` supprime toutes les fenêtres qui ont été ouvertes à l'exception de celle ou se trouve le curseur

3.1.2.9 Sauvegarder et quitter `emacs`

- `C-x C-s` sauvegarde les modifications effectuées si `emacs` connaît le nom de fichier. Sinon l'utilisateur est invité à entrer un nom de fichier dans le mini tampon.
- `C-z` quitte `emacs` provisoirement (le processus est suspendu). Le retour à `emacs` se fait en frappant `fg` ou encore `%emacs`.
- `C-x C-c` quitte `emacs` définitivement.

3.1.3 nedit

`nedit` est un éditeur graphique qui ressemble beaucoup aux éditeurs rencontrés sous les environnements PC, Mac, Amiga (Linux, Windows 3.xx et 9x, DOS, Mac OS, Amiga OS). `nedit` peut être complètement géré à la souris et possède des menus conviviaux pour l'édition, la recherche de texte, le copier coller, la sélection de texte.

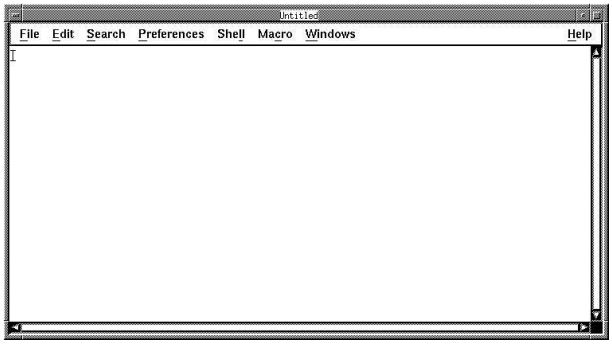


FIG. 3.3 – Fenêtre de l'éditeur graphique `nedit`

Il possède aussi des prédispositions pour la programmation. Il gère l'auto-indentation des lignes de programmes, la gestion du nombre des parenthèses, l'affichage des mots-clés du langage.

Cette page est laissée blanche intentionnellement

Chapitre 4

Unix : Un vrai OS multi-tâches

4.1 Gestion des processus

4.1.1 Introduction

Un processus est un programme binaire en cours d'exécution. Il possède les caractéristiques suivantes :
- identificateur : `PID`
- identificateur du père : `PPID`
- identificateur de l'utilisateur : `UID`
- le répertoire courant
- les fichiers ouverts
- le masque de création : `umask`
- la taille maximum des fichiers créés par le processus : `ulimit`
- la priorité
- les temps d'exécution
- le terminal de contrôle

4.1.2 Obtenir l'`UID` et le `GID`

`id`
Réponse :

Observations :

4.1.3 Visualiser les processus

`ps`
Réponse :

`ps -u`*votre_login*
Réponse :

`ps -aj`
Réponse :

Observations :

4.1.4 Lancer un processus en tâche de fond

nom_du_processus`&`
Lancer `kedit` en tâche de fond
Réponse :

Lancer `kedit`
Réponse :

Observations :

4.1.5 Tuer un processus

`kill -`*nom_signal numéro_de_processus*
Tuer le processus `kedit`
Réponse :

Observations :

4.1.6 Bloquer un processus

Si vous avez lancer un processus long en oubliant de le placer en tâche de fond, il est intéressant de pouvoir le bloquer afin de "reprendre la main" et de le relancer en tâche de fond.
Frapper `<CTRL-Z>`, stoppe le processus en cours et vous "rend la main", il vous est alors possible de la placer en tâche de fond.

4.1.7 Passer un processus en arrière plan

`bg` *numéro_de_job* ou `bg`
Lancer `nedit`, bloquer le, et passer le en tâche de fond
Réponse :

Observations :

4.1.8 Lancer une commande et afficher les temps consommés

`time` *commande*
Lancer une recherche sur les fichiers commençant par `X` dans toute l'arborescence et afficher les temps consommés.
Réponse :

Observations :

Cette page est laissée blanche intentionnellement

Chapitre 5

Vous serez bientôt des gourous UNIX

5.1 Redirections des entrées/sorties, tubes et filtres

5.1.1 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

5.1.2 Créer un fichier contenant la liste des utilisateurs connectés

Réponse :

Vérification :

5.1.3 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)*.

5.1.4 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 :

5.1.5 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 6

$SHELL : csh répondit l'echo

6.1 Le Shell

6.1.1 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`.

6.1.2 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.

6.1.3 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
$
```

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

Réponse :

Vérification :

6.1.4 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
$
```

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

Réponse :

Vérification :

6.1.5 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
$
```

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

Réponse :

Vérification :

6.1.6 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.

6.1.7 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.

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

Réponse :

Observations :

6.1.8 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`.

6.1.9 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**.

6.1.10 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.

6.1.10.1 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.

6.1.10.2 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.

6.1.10.3 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 :

6.1.10.4 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 :

Chapitre 7

Script, programmation et autres douceurs

7.1 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.

7.2 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.

7.2.1 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` $chaîne_1$ `!=` $chaîne_2$

7.2.2 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")

7.2.3 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*.

7.2.4 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).

7.2.5 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 :

7.3 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 :

7.4 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 :

7.5 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 :

7.6 Les itérations non bornées

7.6.1 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 :

7.6.2 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 :

7.7 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 :

7.8 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 $
```

7.9 Écriture de script

7.9.1 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

7.9.2 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
************************************
```

7.9.3 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.

7.9.4 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.

7.9.5 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.
**************************************************
```

7.9.6 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
**************************************************
```

7.9.7 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

7.9.8 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.

7.9.9 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.

7.10 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
```

7.10.1 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
```

Chapitre 8

Filtrer et traîter les chaînes de caractères

8.1 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`.

8.1.1 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.

8.1.1.1 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 :

8.1.1.2 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 :

8.1.1.3 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 :

8.1.1.4 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 :

8.1.1.5 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\>'`.

8.1.2 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
```

8.1.3 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.

8.1.4 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
$
```

8.1.5 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
$
```

8.1.6 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 :

8.2 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.

8.2.1 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 :

8.2.2 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'a*dresse* 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 :

Cette page est laissée blanche intentionnellement

Chapitre 9

Unix Avancé : `find` `cron` et `make`

9.1 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.

9.1.1 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).

9.1.2 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.

9.1.3 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*.

9.1.4 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 :

9.2 Commandes retardées : `at` et `crontab`

9.2.1 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`.

9.2.2 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 {} \;`

9.2.3 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 :

9.3 Automatiser les tâches : `make`

9.3.1 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.

9.3.2 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 :

9.3.3 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.

9.3.4 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 :

9.3.5 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`.

9.3.6 `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 :

Cette page est laissée blanche intentionnellement

Chapitre 10

Introduction à la programmation `Perl`

10.1 Introduction

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*.Perl est l'abréviation de *Practical Extraction and Report Langage* (langage pratique d'extraction et de génération de rapport), même s'il a été également baptisé *Pathologically Eclectic Rubbish Lister* (énumérateur de bétises pathologiquement éclectique).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`.

10.2 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.

10.3 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'
```

10.4 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.

10.4.1 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";'42yann@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.$j}; print "\n";'2yann@yoda:~/script_linux$
```

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

10.4.1.1 Nombres

En Perl, tous les nombres en interne utilisent le même format. Il est possible de spécifier des entiers ou des nombres en virgule flottante (réels, décimaux). Mais en interne Perl ne calcule que des valeurs en virgules flottantes. Il n'y a donc pas de valeurs entières interne à Perl.

Décimaux Le point permet de délimiter les décimaux.

```
1.25
255.000
7.25e45
-6.24e-24
```

Entiers

```
0
2004
-40
61298040283768
61_298_040_283_768
```

Entiers non décimaux Perl vous permet d'indiquer des nombres dans une base différente de la base 10. Les octaux (base 8) commencent par `0`, les héxadécimaux (base 16) commencent par `0x`, les binaires (base 2) commencent par `0b`.

```
perl -e '$toto=0b10 + 0b01;print
$toto\n";'
```

```
0377
0xff
0b11111111
0x50_65_72_7C
```

Opérateurs sur les nombres Perl propose les opérateurs ordinaires d'addition, de soustraction, de multiplication et de division, etc. Par exemple :

```
2 + 3
5.1 -2.4
3 * 12
14 / 2
10.2 /0.3
10 / 3
```

Perl supporte aussi l'opérateur modulo (`%`). La valeur de l'expression 10 % 3 représente le reste de la division entière 10 par 3 soit 1. Il existe aussi un opérateur de puissance (élévation à la puissance) de type FORTRAN. Il est représenté par une double astérisque, par exemple 2 ** 3 représente 2^3 soit 8.

10.4.1.2 Chaînes

Les chaînes sont des suites de caractères (comme `hello`). Elles peuvent contenir n'importe quelle combinaison de n'importe quels caractères. La chaîne la plus courte ne contient qu'un caractère. la chaîne la plus longue replit toute votre mémoire disponible. Les chaînes typiques sont des séquences de caractères imprimables de lettres de chiffres et de signes de ponctuation.

Chaînes entre apostrophes simples Une chaîne entre apostrophes simples est une suite de caractères encadrée par des apostrophes simples `'toto'`. Les apostrophes ne font pas partie de la chaîne, elles ne sont là que pour indiquer à Perl le début et la fin de la chaîne.Tout autre caractère autre que l'apostrophe ou une barre oblique est valide. Pour mettre une barre oblique inverse, il faut la faire précéder d'une autre barre oblique. Pour mettre une apostrophe, il faut aussi la faire précéder d'une barre oblique.

```
'fred''''Ne mettez pas qu\'apostrophe cette chaîne''barre
oblique \\''hello\n'
'hello
there'
'\'\\'
```

dans ce cas le `\n` n'est pas interpété.

Chaînes entre guillemets Une chaînes entre guillemets se comporte comme une chaîne d'autres langages. Il s'agit d'une série de caractères encadrée par des guillemets. Dans ce cas la barre oblique remplt son rôle en indiquant certains caractères de contrôle.

```
"barney"
"hello world\n"
"caractère guillemet \""
"caractère \t tabulation"
```

Contenu	Signification
`\n`	Nouvelle ligne
`\r`	Retour chariot
`\t`	Tabulation
`\f`	Saut de page
`\b`	Espace arrière
`\a`	Bell (sonnerie)
`\e`	Escape (caractère escape ASCII)
`\007`	Toute valeur ASCII Octal (ici 007 bell)
`\x7f`	Toute valeur ASCII Héxadécimal (ici 7f delete)
`\cC`	Toute caractère de controle (ici CTRL-C)
`\\`	Barre oblique inverse
`\"`	Guillemet
`\l`	Lettre suivante en minuscule
`\L`	Lettres suivantes en minuscule jusqu `\E`
`\u`	Lettre suivante en majuscule
`\U`	Lettres suivantes en majuscule jusqu `\E`
`\Q`	Tous les signes non alphanumériques sont transformés en caractères d'échappement jusque `\E`
`\E`	Met fin à `\L` `\U` ou `\Q`

TAB. 10.1 – Tableau récapitulatif des caractères de contrôle en Perl

Les chaînes en guillements sont à variables interpolées, ce qui signifie que les noms de certaines variables situées dans les chaînes sont remplacées par leurs valeurs courantes au moment où les chaînes sont utilisées.

Opérateurs de chaînes Les valeurs d'une chaîne peuvent être concaténées par l'opérateur `.`.

```
"hello" . "world" #équivaut à "helloworld"
"hello" . ' ' . "world" #équivaut à "hello world"
'hello world' . "\n" #équivaut à "hello world\n"
```

l'opérateur répétition de chaîne, constitué par la lettre minuscule `x`. Il prend son opérande de gauche (une chaîne) et en effectue autant de copies concaténées que le spécifie son opérande de droite (un nombre).

```
"fred" x 3 #équivaut à "fredfredfred""barney" x (4+1) #équivaut
à "barneybarneybarneybarneybarney"5 x 4 #équivaut à "5" x 4 soit "5555"
```

Conversion automatique entre nombres et chaînes Perl effectue la conversion automatique des nombres en chaînes si nécessaire. Ceci dépend de l'opérateur utilisé sur la valeur scalaire. Quand un opérateur attend un nombre `+`, Perl voit la valeur comme un nombre. Lorsqu'un opérateur attend une chaîne `.`, Perl voit la valeur comme une chaîne.

```
"Z" . 5 * 7 #équivaut à "Z" . 35 ou "Z35"
```

10.4.1.3 Variables scalaires

Une variable scalaire contient une seule valeur scalaire. Le nom des variables scalaire commence par le caractère `$` suivi d'un identificateur Perl. Cette identificateur est constitués de caractère alphanumérique mais ne peut pas être commencé par un chiffre. De plus on distingue les majuscules des minuscules. Par emple la variable `$Fred` est différente de `$fred`.
Afin de bien nommer ses variables, il faut leur donner un nom explicite ni trop court ni trop long.
Une variable qui sera utilisée seulement dans deux ou trois lignes rapprochées portera un nom simple, `$n` par exemple, alors qu'une variable utilisée dans tout un programme aura un nom plus significatif.
La plupart du temps les noms de variable de vos programmes seront en minuscules. Dans certains cas particulier on utilise des majuscules. Le tout majuscule (comme `$ARGV`) indique généralement que cette variable comporte des caractéristiques spéciales.
L'affectation des variables scalaires est l'opération la plus souvent utilisées. L'opérateur d'affectation en Perl est la signe `=` qui prend un nom de variable à gauche et lui affecte la valeur de l'expression située à droite.

```
$fred = 17;
$barney = 'hello';
$barney = $fred +3;
```

Il existe aussi des opérateurs d'affectation binaires qui sont en sorte des raccourcis d'opérations plus affectation. Ces opérateurs permettent de réduire la quatité de code.

```
$fred = $fred + 5;
$fred += 5;
```

Les deux affectations précédentes sont équivalentes. On peut ainsi écrire un opérateur d'élévation à la puissance `**=`

10.4.1.4 Sortie avec `print`

IL est souvent préférable d'imprimer les résultats. C'est l'opérateur `print` qui permet cela. Il prend en argument un scalaire et le place sur la sortie standard. Normalement il 's'agit de l'écran de votre terminal.

```
print "hello world\n";
print "la réponse est ";
print 6 * 7;
print "\n";
```

On peut aussi fournir une série de valeurs séparées par des virgules

```
print "la réponse est ", 6 * 7, ".\n";
```

10.4.1.5 Interpolation de variables scalaires en chaînes

Lorsqu'une chaîne est entre guillemets, elle est sujette à l'interpolation de variables. Cela signifie que tout nom de variable figurant dans la chaîne est remplacé par sa valeur en cours.

```
$meal = "frites dans la purée\n"
$plat = "la fondue belge c'est des $meal";
```

On désigne l'interpolation de variable par interpolation par guillements. Pour placer un vrai `$` il faut le faire précéder d'un `\`.
Si vous voulez accoler des lettres directements à la suite du contenu d'une variable, il faut utiliser les délimiteurs de noms de variables que sont les accolades.

```
$fruit = "pomme";
$phrase = "j\'ai 5 ${fruit}s\n;
```

10.4.1.6 Précédence et associativité des opérateurs

La précédence des opérateur permet de déterminer les cas ambigus dans lesquels deux opérateurs doivent intervenir sur 3 opérandes. Par exemple dans l'opération `2+3*5` le résultat sera `25` ou `17`.
il faut donc définir une précédence des opérateurs pour lever l'ambiguité. Il est possible de passer cette précédence en plaçant des parenthèses au endroits désirés.

Associativité	Opérateurs
gauche	les parenthèses et les opérateurs de listes
gauche	`->` `++ --` (auto-incrémentation, auto-décrémentation)
droite	`**`
droite	`\ ! ~ + -` opérateurs uniaires
gauche	`= ~ !~`
gauche	`* / % x`
gauche	`+ - .` opérateurs binaires
gauche	`<< >>` opérateurs unaires nommés (tests de fichiers -X, rand) `< <= > >= lt le gt ge` (les "non egal") `== != <=> eq ne cmp` (les "égal")
gauche	`&`
gauche	`| ^`
gauche	`&&`
gauche	`||`
droite	`? :` (ternaire)
droite	`= += -= .=` (et les opérateurs d'affectations similaires)
gauche	`, =>` (opérateurs de listes vers la droite
droite	`not` (non logique)
gauche	`and` (et logique)
gauche	`or` (ou logique) `xor` (ou exclusif)

TAB. 10.2 – Associativité et précédence des opérateurs du plus élevé au plus faible

10.4.1.7 Opérateurs de comparaison

Pour comparer les nombres, Perl propose les opérateurs de comparaison logique décrit dans le tableau ci-dessus. Ils renvoient *true* vrai ou *false* faux.

Comparaison	Numérique	Chaîne
Égal à	`==`	`eq`
Différent de	`!=`	`ne`
Inférieur à	`<`	`lt`
Supérieur à	`>`	`gt`
Inférieur ou égal à	`<=`	`le`
Supérieur ou égal à	`>=`	`ge`

TAB. 10.3 – Opérateurs numériques et de comparaison de chaînes

10.4.1.8 Valeurs booléennes

Contrairement à certains langages, Perl ne possède pas de type booléen indépendant, mais respecte quelques règles simples :

1. la valeur spécial `undef` est fausse ;
2. zéro est faux ; tous les autres nombres sont vrais ;
3. la chaîne vide `''` est fausse ; toutes les autres chaînes sont normalement vraies ;
4. seule exception, dans la mesure où les nombres et les chaînes sont équivalents, la forme chaîne de zéro `'0'` possède la même valeur que sa forme numérique fausse.

10.4.1.9 `STDIN` en tant que variable scalaire

Comment obtenir une valeur saisie au clavier dans un programme Perl. Le moyen le plus simple est l'opérateur d'entrée standard `<STDIN>`. Lorsque vous utilisez `<STDIN>` à un endroit où une valeur scalaire est attendue, Perl lit la totalité de la ligne de texte suivante de l'*entrée standard* jusqu'au prochain caractère de nouvelle ligne, et utilise cette chaîne comme valeur de `<STDIN>`.

```
$line = <STDIN>;
if ($line eq "\n")
{
print "Ceci n'était qu'une ligne vide!\n";
}
else
{
print "Cette ligne de saisie était : $line";
}
```

10.4.1.10 L'opérateur `chomp`

Cet opérateur enlève le dernier caractère d'une variable si celui-ci est un caractère de nouvelle ligne.

```
$text = "une ligne de texte\n";
chomp($text);
```

Il est alors possible de faire directement une affectation dans une variable à partir de `STDIN` en enlevant en même temps le caractère de fin de ligne.

```
chomp($text=<STDIN>);
```

```
$text = <STDIN>;
chomp($text);
```

`chomp` est une fonction et renvoie une valeur : le nombre de caractères supprimés (ce qui est rarement utile). Lorqu'une ligne se termine par plusieurs caractère de fin de ligne, la fonction `chomp` n'en supprime qu'un.

10.4.1.11 La valeur `undef`

Que se passe-t-il si vous utilisez une variable scalaire avant de l'avoir affecté. Rien de fatal, les variables possèdent une valeur spéciale `undef` avant d'avoir reçu leur première affectation. Si on essaie d'utiliser `undef` comme un nombre, la valeur vaut 0, comme un chaîne, la valeur vaut la chaîne vide.
Il est possible aussi de fixer une valeur à `undef`.

```
$brel = undef;
```

10.4.1.12 La fonction `defined`

L'opérateur d'entrée standard `<SDTIN>` peut renvoyer `undef` s'il n'y a plus d'entrée (fin de fichier par exemple). Pour savoir si une valeur est `undef`, on utilise la fonction `defined`, qui renvoie faux pour `undef` et vrai pour tout le reste.

```
$brel = <STDIN>;
if (defined($brel))
{
print "La saisie est $brel";
}
else
{
print "Aucune saisie disponible!\n";
}
```

10.4.2 Listes et tableaux

Une liste représente un ensemble de données scalaires ordonnées. Un tableau est une variable contenant une liste. Les termes sont souvent confondus, mais la liste représente les données et le tableau la variable. Une valeur de liste ne peut pas se trouver dans un tableau, mais toute variable tableau contient une liste.
Chaque élément de tableau ou de liste est une variable scalaire indivuduelle comportant une valeur scalaire indépendante.
Ces valeurs sont indexées par des entiers à partir de zéro. Le premier élément d'une liste est l'élément zéro.
Un tableau peut donc contenir une valeur scalaire nombre, chaîne `undef`. Le plus souvent on utilse des tableaux de même type.

10.4.2.1 Accès aux éléments d'un tableau

Perl propose une fonction d'indiçage permettant de faire référence à un élément du tableau par un indice numérique. Les éléments d'un tableau sont numérotés par une suite d'entiers s'incrémentant à partir de zéro.

```
$toto[0] = "titi";
$toto[1] = "tutu";
$toto[2] = "tata";
```

le nom du tableau `toto` provient d'un espace de nom complètement indépendant de celui utilisé par les scalaires. On peut donc avoir une variable scalaire `toto` dans le même programme ; Perl les traiteras comme des entités différentes sans les confondre.
Il est possible d'employé un élément de tableau tel que `toto[2]` à quasiment tout endroit du programme ou il est possible d'utiliser une variable scalaire (sauf dans les contrôle de boucle). Il est possible modifier la valeur d'un élément d'un tableau.

```
print $toto[0];
$toto[1] = "tututiti";
$toto[2] = "tatatiti";
```

Bien sur il est possible de donner une expression donnant un valeur numérique en guise d'indice de tableau. Si cet valeur n'est pas un entier, elle est automatiquement tronquée à l'entier inférieur le plus proche.

```
$nombre = 1.725;
print $toto[$nombre - 1]; #identique à print $toto[0];
```

si l'indice d'un élément est au delà de la fin du tableau, la valeur correspondante est `undef` comme dans le cas des scalaires ordinaires. Quand une variable n'a jamais stockée de valeur elle est `undef`.

```
$blanc = $toto[127_234];
$blanc = $mel;
```

10.4.2.2 Indices spéciaux d'un tableau

Si vous effectuer un rangement dans un élément situé au delà de la fin d'un tableau ce dernier est automatiquement étendu. En cas de besoin les éléments intermédiaires sont créés en tant que valeur `undef`.

```
$roches[0] = 'sediment';
$roches[1] = 'ardoise';
$roches[2] = 'lave';
$roches[3] = 'minerai';
$roches[99] = 'shiste';
```

On a parfois besoin de l'indice du dernier élément du tableau. Pour le tableau `roches`, il s'agit de la valeur `$#roches`. Elle n'est pas égale au nombre d'élément, car il existe un élément zéro. Il est aussi possible d'affecter cette valeur pour modifier la taille du tableau.

```
$fin = $#roches; # 99, indice du dernier élément
$nombre_de_roches = $fin +1; # 100 éléments
$#roches = 2; # ouble toutes les roches après lave
$#roches = 99; #ajoute 97 élément undef
$roches[$#roches] = 'roches dures'; #la dernière roche
```

Il est assez courant d'utiliser la valeur `$#name` comme indice. Les indices de tableau négatifs comptent à partir de la fin du tableau. Mais ces indices ne bouclent pas. Si un tableau comportent 3 éléments, les indices négatifs sont `-1` (dernier élément), `-2` (l'élément intermédiaire) et `-3` (le premier élément).

```
$roches[-1] = 'roche dure'; # affectation du dernier élément
$dead_rock = $roches[-100]; # donne sédiment
$roches[-200] = 'cristal'; # erreur fatale !
```

10.4.2.3 Littéraux de liste

Un littérale de liste (manière dont vous représentez la valeur d'une liste à l'intérieur de votre programme), est constitués de valeurs séparées par des virgules, entre parenthèses. Ces valeurs constituent les éléments de la liste.

```
(1,2,3) #liste de 3 valeurs
(1, 2, 3,) #liste de 3 valeurs virgule de terminaison
("fred", 4.5) # 2 valeurs
() # liste vide
(1..100) #liste de 100 entiers
```

La dernière utilise l'opérateur étendue `..`, il crée une liste de valeurs en dénombrant à partir du scalaire de gauche jusqu'au sclaire de droite, une par une.

```
(1..5) #identique à (1,2,3,4,5)
(1.7..5.7) # idem mais les valeurs sont tronquées
(5..1) # liste vide ne dénombre qu'en remontant
(0, 2..6, 10, 12) # identique à (0,2,3,4,5,6,10,12)
($a..$b) # étendue déterminée par des valeurs
(0..$#roches) # les indices du tableau de roches de la section précédente
```

Les élément d'un tableau ne sont pas forcément des constantes. Ils peuvent être des expressions réevaluées à chaque utilisqation du littéral.

```
($a, 17) #deux valeurs : la valeur actuelle de $a et 17
($b+$c, $d+$e) #deux valeurs
```

On peut aussi avoire une liste de chaîne.

```
("toto", "tutu", "titi", "tata")
```

10.4.2.4 Le raccourcis `qw`

On a très souvent recours aux listes de mots simples. Le raccourci `qw` facilite leur génération en épargnant la saisie de nombreuses guillemets :

```
qw/ toto tutu titi tata/ #idem ci dessus mais moins de lettres frappées
```

`qw` signifie *quoted words* (mots entre guillemets) ou *quoted with whitespace* (cités par des blancs). Perl les traîte commeune chaîne entre apostrophes (on ne peut donc pas utiliser `\n` ou `$toto` dans une liste `qw`). Le blanc (espace ou nouvelle ligne) sont écartés.

```
qw/ toto
tutu titi
tata/ #idem ci dessus mais moins de lettres frappées
```

De même il n'est pas possible d'insérer des commantaires dans une liste `qw`. Perl permet de choisir n'importe quel délimiteur de `qw`. Tous les caractères de ponctuation peuvent être utilisé.

```
qw/ toto tutu titi tata/
qw! toto tutu titi tata!
qw{ toto tutu titi tata}
qw( toto tutu titi tata)
qw[ toto tutu titi tata]
qw< toto tutu titi tata>
```

Si vous désirez toujours utiliser le même délimiteur et insérer ce caractère dans la liste, il est toujours possible de la faire en le faisant précéder d'un `\`.

```
qw! yahoo\! google excite lycos! #comprend l'élément yahoo!
```

10.4.2.5 Affectation de liste

Comme les valeurs scalaires, il est possible d'affecter des valeurs de listes à des variables.

```
($toto, $barney, $dino) = ("silex", "ruine", undef);
```

les trois variables de la liste de gauche obtiennent de nouvelles valeurs, comme si l'on avait fait 3 affectations séparées. La liste étant constituée avant l'affectation, il est facile d'échanger deux variables en Perl :

```
($toto, $barney) = ($barney, $toto); # permute ces valeurs
($betty[0], $betty[1]) = ($betty[1], $betty[0]);
```

Que se passe t'il si le nombre de variables à gauche est différents du nombre de valeurs à droites. dans l'affectation d'une liste, les valeurs en trop sont silencieusement ignorées. Par contre si vous avez trop de variables, les extras obtiennent la valeur `undef`.

```
($toto, $barney) = qw < silex ruine ardoise granite >; # deux éléments ignorés
($wilma, $dino) = qw[silex]; # $dino reçoit undef
```

Voici maintenant un tableau de chaînes en un ligne :

```
($roche[0], $roche[1], $roche[2], $roche[3]) = qw / talc mica feldspath quartz /;
```

Mais pour faire référence à un tableau de scalaire, Perl propose une notation simple le signe `@` devant le nom du tableau. On peut le lire comme "l'intégralité de", et par conséquent `@roches` signifie l'intégralité de roches. Ceci fonctionne des deux côtés de l'opérateur d'affectation.

```
$roches = qw/ sediment ardoise lave /;
@petit = (); #liste vide
@gigantesque = 1..1e5; # liste de 100 000 éléments
@substance = (@gigantesque, undef, @gigantesque); # liste de 200 001 éléments
$dino = "granite";
@carriere = (@roches, "minerai", @petit, $dino);
```

Cette dernière affectation procure à `@carriere` la liste de 5 éléments `(sediment, ardoise, lave, minerai, granite)`, car `@petit` ne contribue pas.
Un nom de tableau est remplacé par la liste qu'il contient, il ne devient pas un élément de la liste, car ces tableaux ne peuvent contenir que des scalaires et pas d'autres tableaux (pour le moment).
La valeur d'une variable tableau non affectée est `()`, la liste vide.
Lorsqu'un tableau est recopié dans un autre, cela demeure une affectation de liste. Les listes sont simplement stockées dans un tableau.

```
@copie = @carriere; #copie d'une liste d'un tableau à un autre
```

10.4.2.6 Les opérateur `push` et `pop`

Il est possible d'ajouter de nouveaux éléments à la fin d'un tableau en les rangeant avec des indices plus élevés. Mais les programmeurs Perl utilisent les tableaux comme des empilements d'informations dans lesquels les nouvelles valeurs sont ajoutées ou enlevées du côté droit de la liste (les derniers éléments de la liste).
L'opérateur `pop` enlève le dernier élément d'un tableau,puis il le renvoie :

```
@tableau = 5..9;
$toto = pop(@tableau); # $toto reçoit 9, @tableau contient maintenant (5,6,7,8)
$barney = pop(@tableau); # $barney reçoit 8, @tableau contient maintenant (5,6,7)
pop @tableau; # @tableau contient maintenant (5,6)
```

Le dernier exemple utilise `pop` dans le contexte vide.
Lorsque le tableau est vide, `pop` ne s'en occupe pas et renvoie `undef`. `pop` peut être utiliser avec ou sans parenthèses, ceci est une règle générale en Perl.
L'opérateur inverse est `push`, elle ajoute un élément (ou une liste d'éléments) à la fin d'un tableau :

```
push(@tableau, 0); #@tableau contient maintenant (5,6,0)
push @tableau, 8; #@tableau contient maintenant (5,6,0,8)
push @tableau, 1..10; #@tableau contient maintenant (5,6,0,8,1,2...10)
@autres = qw / 9 0 2 1 0 /;
push @tableau, @autres; # @tableau contient ces 5 nouveaux éléments (19 au total)
```

Le premier argument de `push` et l'argument de `pop` doit être une variable tableau.

10.4.2.7 Les opérateurs `shift` et `unshift`

Les opérateurs `push` et `pop` agissent sur la fin d'un tableau. Les opérateurs `unshift` et `shift` effectuent une opération correspondante au début du tableau (côté gauche) :

```
@tableau = qw # dino toto barney #;
$a = shift(@tableau); #$a reçoit "dino" et tableau contient toto barney
$b = shift @tableau; #$b reçoit "toto" et tableau contient barney
shift @tableau; #@tableau est vide
$c = shift @tableau; #$c reçoit undef, tableau est vide
unshift(@tableau, 5); #@tableau contient (5)
unshift @tableau, 4; #@tableau contient (5,4)
@autres = 1..3;
unshift @tableau, @autres; # tableau contient (1,2,3,4,5)
```

De manière analogue à `pop`, `shift` renvoie `undef` en cas de variable de tableau vide.

10.4.2.8 Interpolation de tableau en chaînes

Comme les scalaires, les valeurs des tableaux peuvent être interpolées dans les chaînes entre guillemets. Les éléments du tableau sont automatiquement séparés par des espaces :

```
@roches = qw{ silex ardoise ruine };
print "quartz @roches calcaire\n"; # affiche 5 roches séparées par des espaces
```

Aucun espace supplémentaire n'est ajouté avant ou après un tableau interpolé. Le cas échéans vous devez les placer vous même :

```
print "voici trois roches : @roches.\n";
print "Ici les parenthèses (@empty) ne contiennent rien.\n";
```

Attention à cette manière d'interpoler lorsque vous voulez écrire des adresses email.

```
$email = "toto@sediment.edu" #erreur essaie d'interpoler @sediment
$email = "toto\@sediment.edu"; # Correct
$email = 'toto@sediment.edu'; # Correct
```

Un élément individuel d'un tableau sera remplacé par sa valeur.

```
@toto = qw (hello dolly);
$y = 2;
$x = "Voici la place de $toto[1]"; # Voici la place de dolly
$x = "Voici la place de $toto[y-1]"; #idem
```

L'index est calculé comme une expression ordinaire, comme si elle était extérieure à la chaîne. Elle n'est pas d'abord interpolée comme dans une variable. Si `$y` contient la chaîne "2*4", nous parlons encore de l'élément 1, car la `"2*4"` en tant que nombre vaut 2.
Si vous voulez faire suivre une variable scalaire d'un crochet gauche, vous devez délimiter celui-ci afin qu'il ne soit pas considéré comme faisant partie d'une référence de tableau :

```
@toto = qw(manger roches est mauvais);
$toto = "bon";
print "Cela est $toto[3]\n";# affiche "mauvais"
print "Cela est ${toto}[3]\n"; #affiche "bon[3]"
print "Cela est $toto"."[3]\n"; # encore bon
print "Cela est $toto\[3\]\n"; # encore bon
```

10.4.2.9 Structure de contrôle `foreach`

Il est souvent commode de pouvoir traiter tous les éléments d'un tableau ou d'une liste. Perl propose une structure de contrôle à cette fin.La boucle `foreach` parcours une liste de valeurs en exécutant une itération pour chaque valeur :

```
foreach $roches (qw/ sediment ardoise lave /)
{
print "Une roche est $roches.\n";
}
```

```
perl -e 'foreach $roches (qw/ sediment ardoise lave /){print "Une roche est $roches.\n";}'
Une roche est sediment.
Une roche est ardoise.
Une roche est lave.
```

La variable de contrôle `$roches` prend une nouvelle valeur de la liste à chaque itération. La variable de contrôle n'est pas une copie de l'élément de la liste, c'est l'élément de la liste. Si la variable de contrôle est modifiée, l'élément de la liste est modifié.

```
@roches = qw/ sediment ardoise lave /;
foreach $roche (@roches)
{
$roche = "\t$roche"; #place une tabulation devant chaque élément
$roche .= "\n"; # place une nouvelle ligne à la fin de chaque élément
}
print "les roches sont :\n", @roches;
```

```
perl -e '@roches = qw/ sediment ardoise lave /;foreach $roche (@roches){$roche = "\t$roche";$roche .= "\n"
les roches sont :
        sediment
        ardoise
        lave
```

Après la boucle, la variable de contrôle est identique à la valeur avant la boucle. Cette valeur est sauvegardée par Perl, mais nous n'y avons pas accès.

10.4.2.10 La variable par défaut :$_

Si vous oubliez la variable de contrôle au début d'une boucle `foreach`, Perl utilise sa valeur par défaut préférée `$_`. Elle ressemble à n'importe quelle variable scalaire.

```
foreach (1..10)
{
print "je sais compter jusqu'à $_!\n";
}
```

C'est la valeur la plus courante par défaut. Perl utilisera souvent `$_` lorsque vous ne lui indiquez aucun nom de variable.

```
$_ = "zorro est arrivé\n";
print ;
```

```
perl -e '$_ = "zorro est arrivé\n";print;'
zorro est arrivé
```

10.4.2.11 L'opérateur `reverse`

Cet opérateur prend une liste de valeurs et renvoie la liste dans l'ordre opposé.

```
@toto = 6..10;
@barney = reverse(@fred); # reçoit 10 9 8 7 6
@wilma = reverse 6..10;#idem
@toto = reverse @toto; remplace la liste d'origine
```

Attention, `renverse` renvoie la liste inversée, elle ne touche pas à ses arguments.

```
reverse @toto; #Mauvais ne modifie rien.
@toto = reverse @toto;
```

10.4.2.12 L'opérateur `sort`

Cet opérateur prend une liste de valeurs et les trie par ordre alphabétique. Pour les chaînes ASCII, il s'agit de l'ordre ASCII.

```
@roches = qw/ sediment ardoise ruine granite/;
@triee = sort(@roches);
@inverse = reverse sort @roches;
```

```
perl -e '@roches = qw/ sediment ardoise ruine granite/;@triee =
sort(@roches);@inverse = reverse sort @roches;print "@triee\n"; print
"@inverse\n";'
ardoise granite ruine sediment
sediment ruine granite ardoise
```

Comme pour `reverse`, il faut un opérande de gauche pour pouvoir utiliser la fonction.

10.4.2.13 Contexte de liste et contexte scalaire

Cette section est très importante.
L'idée est simple : la signification d'une expression donnée peut être différente selon l'endroit où elle apparaît. Comme dans la langue orale ou écrite, une expression dépend du `contexte`.
Le `contexte` fait référence à un endroit où figure une expression. Au fur et à mesure que Perl analyse les axpressions il attend une valeur scalaire ou une valeur de liste. Perl attend le contexte de l'espression.

```
5 + qquechose; #qquechose doit être scalaire
sort qquechose; # qquechose doit être liste
```

Même si `qquechose` est exactement la même séquence de caractères, dans un cas il peut donner une valeur scalaire unique, et dans un autre cas, une liste.
Les expressions Perl renvoient toujours la valeur appropriée à leur contexte. Par exemple, un nom de tableau, dans un contexte de liste renvoie la liste de des ses éléments. Dans un contexte scalaire il renvoie le nombre d'éléments du tableau.

```
@personnes = qw ( toto barney betty);
@triee = sort @personnes; #contexte de liste : barney betty toto
$nombre = 5 + @personnes; #contexte scalaire : 5 + 3 => 8
```

Même une affectation ordinaire entraîne des contextes différents :

```
@liste = @personnes; #liste de 3 personnes
$n = @personnes; #le nombre de personnes : 3
```

10.4.2.14 Utilisation d'expressions produisant une liste dans un contexte scalaire

De nombreuses expressions sont susceptibles de produire des listes. Qu'obtient on si on utilise une liste dans un contexte scalaire.
Certaines expressions ne possèdent aucune valeur de contexte scalaire. Par exemple une commande `sort` sur un contexte scalaire renverra `undef`.
La commande `reverse`, dans un contexte de liste renvoie une liste inversée. Dans un contexte scalaire, elle renvoie une chaîne inversée (inverse le résultat de la concaténation de toutes les chaînes de la liste).

```
@inverse=reverse qw/ yabba dabba doo/; # donne doo dabba yabba
$inverse=reverse qw/ yabba dabba doo/; # donne oodabbadabbay
```

Voici quelques contextes courant pour vous familiariser avec les contextes :

```
$fred = quelquechose; #contexte scalaire
@cailloux = quelquechose; #contexte de liste
($wilma, $betty) = quelquechose; #contexte de liste
($dino) = quelquechose; #contexte de liste
```

Voici quelques contextes scalaires :

```
$fred = quelquechose;
$fred[3] = quelquechose;
123 + quelquechose;
quelquechose + 654;
if (quelquechose) { ... }
while (quelquechose) { ... }
$fred[quelquechose] = quelquechose;
```

Voici quelques contextes de liste :

```
@toto = quelquechose;
($toto,$barney) = quelquechose;
($toto) = quelquechose;
push @fred, quelquechose;
foreach $fred (quelquechose) { ... }
sort quelquechose;
reverse quelquechose;
print quelquechose;
```

10.4.2.15 Utilisation d'expressions produisant un scalaire dans un contexte de liste

Dans ce cas les choses sont simples : si une expression ne possède pas normalement une valeur de liste, la valeur scalaire est automatiquement promue pour créer une liste à un élément :

```
@toto = 6 * 7; # reciot la liste à un élément (42)
@barney = "hello" . ' ' . "world";
```

Attention, il y a un piège :

```
@wilma = undef; # On obtient une liste à un élément (undef)
# qui n'est pa la même que celle-ci
@betty = (); # Facon correcte de vider un tableau
```

10.4.2.16 Imposer un contexte scalaire

On a parfois besoin d'imposer un contexte scalaire là où Perl attend la liste. Vous pourrez alors utiliser la fonction factice `scalar`. Il ne s'agit pas d'une véritable fonction, elle indique simplement à Perl de fournir un contexte scalaire :

```
@roches = qw( talc quartz jade obsidienne);
print "combien avez vous de roches ?\n";
print "J'ai", @roches, " roches!\n"; #Erreur affiche les noms des roches
print "J'ai", scalar @roches, " roches!\n"; # correct, donne un nombre
```

Par contre il n'existe pas de fonction pour imposer un contexte de liste.

10.4.2.17 <STDIN> dans une contexte de liste

`<STDIN>` renvoie la prochaine ligne en entrée dans un contexte scalaire. Mais dans un contexte de liste, il renvoie toutes les lignes restantes jusqu'à la fin du fichier. Chaque ligne est renvoyée dans un élément individuel de la liste.

```
@lignes = <STDIN>; # lit l'entrée standard dans un contexte de liste
```

Lorsque l'entrée provient d'un fichier, le reste du fichier est lu.
Lorsque l'on fournit une liste de lignes à la fonction `chomp`, celle ci supprime les caractères de nouvelle ligne de tous les éléments de la liste.

```
@lignes = <STDIN>; # lit toutes les lignes
chomp(@lignes); # supprime tous les caractères de nouvelle ligne
```

ou encore :

```
chomp(@lignes = <STDIN>); # lit toutes les lignes et supprime tous les caractères de nouvelle ligne
```

10.5 Les Sous Programmes

10.5.1 Fonctions système et utilisateur

Comme d'autres langage de programmation, Perl peut définir des sous-programmes qui sont des fonctions crées par l'utilisateur. La réutilisation de code est alors possible.
Le nom d'un sous-programme Perl est un identificateur Perl (lettres, chiffres, soulignés, mais ne pas commencer par un chiffre) précédé d'une esperluète (&) parfois facultative.
Le nom de sous-programme provient d'un espace de noms indépendants. Comme cela Perl ne fait pas de confusion en présence d'une fonction `&toto` et d'un scalaire `$toto` dans un même programme.

10.5.2 Définition d'un sous-programme

Pour définir un sous programme, on utilise le mot-clé `sub`, le nom de fonction (sans esperluète), puis le code indenté (entre accolades) qui constitue le coprs de la fonction. Par exemple :

```
sub marine
{
$n+=1; #variable globale $n
print "Hello marin numéro $n!\n";
}
```

La définition des fonctions peuvent être situées à n'importe quel endroit de votre programme. Les définitions des fonctions sont globales ; et en l'absence d'astuces, il n'existe pas de fonctions privées. Si deux fonctions portent le même noms, la dernière supplante la première.
Dans tous les exemples précédents, toutes les variables étaient globales. Il est possibles de créer des variables privées.

10.5.3 Appel d'un sous-programme

On appelle un sous-programme (ou fonction utilisateur) depuis n'importe quelle expression en indiquant son nom (précédé d'une esperluète).

```
&marine; #affiche Hello marin numéro 1!
&marine; #affiche Hello marin numéro 2!
&marine; #affiche Hello marin numéro 3!
&marine; #affiche Hello marin numéro 4!
```

10.5.4 Valeur de retour

Le sous-programme est toujours appellé depuis une expression, même lorsque le résutat de celle-ci n'est pas utilisée. En appelant &marine (ci-dessus), nous avons calculé la valeur de l'expression contenant l'appel, sans tenir compte du résultat.
Le plus souvent, on utilise un sous-programme et on utilise son résultat, c'est à dire la valeur de retour. Tous les sous-programmes Perl comporte une valeur de retour utile ou non.
Les valeurs renv

10.5.5 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";'dimlunmarmerjeuvensamyann@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
$nb_jours;print "\n";'7yann@yoda:~/script_linux$
```

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 $i;print
"\n";'4yann@yoda:~/script_linux$ 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
$semaine;print "\n";'7yann@yoda:~/script_linux$
```

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:".$debut." fin:"
valeurs:".@valeurs."\n";'debut:4 fin: valeurs:4
[yann@ulysse script_linux]$
```

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 :

10.5.6 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 "$t[0]\n"'3[0]1[yann@ulysse
script_linux]$
```

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.plfor ($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]$
```

10.6 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.

10.6.1 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 `+=`, `-=`, `/=`, `*=`.

10.6.2 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 $a . "\n";'abcabcabcdefyann@yoda:~/script_linux$
```

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";}'OKyann@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.

10.7 Structures de contrôles

10.7.1 Structure de test

10.7.1.1 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.

10.7.1.2 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.

10.7.1.3 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);
```

10.7.2 Structures de boucles

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

10.7.2.1 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 $i\n"} while (1 == 0);'
action
[yann@ulysse yann]$
```

10.7.2.2 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));
```

10.7.2.3 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.

10.7.2.4 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]$
```

10.7.2.5 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.

10.7.2.6 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.

10.8 Définitions de fonctions

10.8.1 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);
```

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.

10.8.2 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]$
```

10.8.3 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=$b\n";
efface (12);
[yann@ulysse 14]$
```

10.8.4 Portée des variables

10.8.4.1 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 ?

10.8.4.2 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 ?

10.8.5 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.

10.8.6 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 11

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

11.1 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`

11.2 Création de Processus : `fork`

11.2.1 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.

11.2.2 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 :

11.2.3 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 11.1), relever les numéros des processus et numéroter l'ordre d'exécution des instructions `printf` de façon à retrouver l'ordre d'exécution des processus.

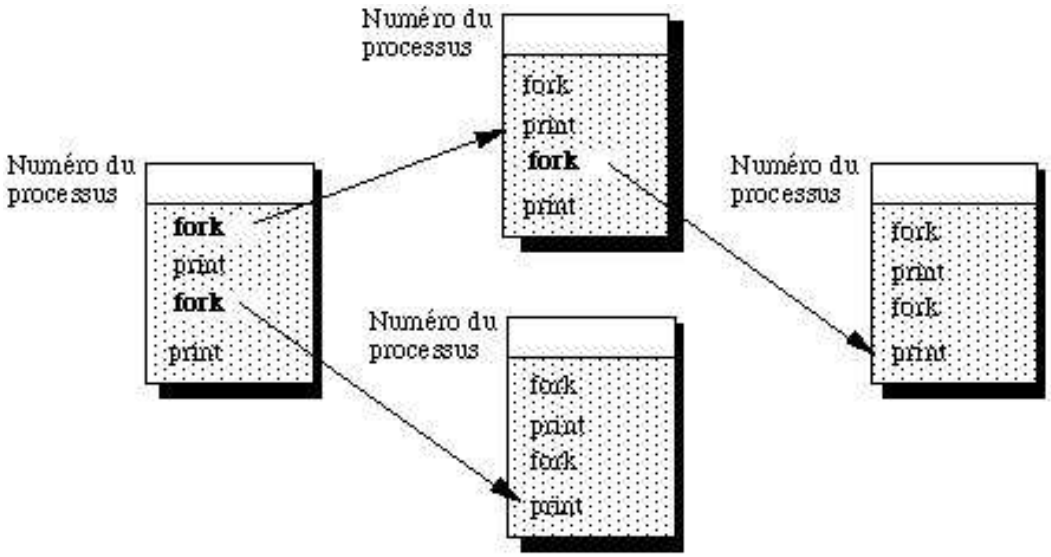


FIG. 11.1 – Hiérarchie des processus

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 :

11.3 Père et fils exécutent des programmes différents

11.3.1 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`.

11.3.2 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.

11.3.3 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 :

11.3.4 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.

11.3.4.1 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 :

11.3.4.2 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 :

11.3.4.3 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 :

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

11.4.1 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 11.2) :

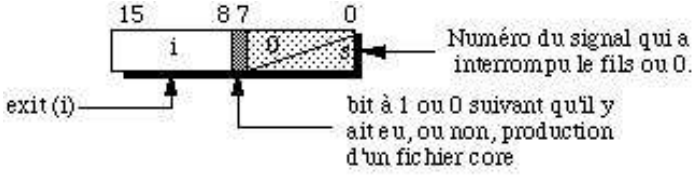


FIG. 11.2 – Retour de la fonction `wait`

11.4.2 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 :

11.4.3 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 :

Chapitre 12

Programmation Multitâche (Threads)

12.1 Introduction

Ce TP est issu d'un article de Pierre Ficheux (`pficheux@com1.fr`). Il s'agit d'une introduction à la programmation multi-threads sous LINUX. Les exemples de programmation utilisent la bibliothèque LinuxThreads disponible en standard sur la majorité des distributions LINUX récentes.

12.2 Qu'est-ce que le multi-threading ?

Les programmeurs LINUX et plus généralement UNIX sont depuis longtemps habitués aux fonctionnalités multi-tâches de leur système préféré. Tous ceux qui se sont frottés un tant soit peu à la programmation système savent qu'il est aisé sous UNIX de créer des processus fils à partir d'un processus existant en utilisant l'appel système `fork`, comme le montre le petit exemple de code ci-dessous : **fork.c**

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

int i;

main (int ac, char **av)
{
  int pid;

  i = 1;

  if ((pid = fork()) == 0) {
    /* Dans le fils */
    printf ("Je suis le fils, pid = %d\n", getpid());
    sleep (2);
    printf ("Fin du fils, i = %d !\n", i);
    exit (0);
  }
  else if (pid > 0) {
    /* Dans le pere */
    printf ("Je suis le pere, pid = %d\n", getpid());
    sleep (1);

    /* Modifie la variable */
    i = 2;
    printf ("le pere a modifie la variable a %d\n", i);

    sleep (3);
    printf ("Fin du pere, i = %d !\n", i);
    exit (0);
  }
```

```
  else {
    /* Erreur */
    perror ("fork");
    exit (1);
  }
}
```

Taper ou récupérer le programme `fork.c`.

Pour le compiler ,utiliser la commande `gcc`, par exemple (premier exercice) : `gcc fork.c -o exo1` ou `gcc -Wall fork.c -o exo1`. Pour plus de détail taper la commande `man gcc`.

Que se passe-t-il ? Pourquoi ? Quelles sont les limites du `fork` ?
Réponse :

la création d'un nouveau contexte est pénalisante au niveau performances. Il en est de même pour le changement de contexte (context switch), lors du passage d'un processus à un autre.
Un *thread* ressemble fortement à un processus fils classique à la différence qu'il partage beaucoup plus de données avec le processus qui l'a créé :
– Les variables globales,
– Les variables statiques locales,
– Les descripteurs de fichiers (file descriptors).
Le multi-threading est donc une technique de programmation permettant de profiter des avantages (et aussi de certaines contraintes) de l'utilisation des *threads*.

12.3 Le multithreading

Cette partie est tirée d'un article d'Éric Lacombe (tuxico@free.fr) du magazine Linux France Magazine du mois de Juillet/Août 2004.
Le multithreading est l'alternative à la programmation multiprocessus. Il offre un parallélisme plus léger à gérer pour le système.
Les threads créent un parallélisme intra-processus et facilitent le partage des tâches et leur communication au sein d'un processus.

12.3.1 Les threads en théorie : Les processus et les threads

Un thread est un fil ou flot d'exécution. Lors de la création d'un processus, un thread est créé : c'est lui qui contiendra les informations nécessaires à l'exécution du programmedont le code se situe dans l'espace d'adressage du processus.
Quand un processus n'est pas multithreadé, il n'a la possibilité que de s'exécuter séquentiellment. Il ne possède donc qu'un seul compteur ordinal , une seule pile de données temporaires, (adresse de retour, variables temporaires...) et une pile système (cas de linux) allouée par le noyau dans son espace d'adressage qui lui sert au passage de paramètres des appels systèmes effectués par le processus appelant.
Un processus contient également une section de données réservée aux variables globales ainsi qu'un *tas* (structure de données arborescente classique) utilisé pour les variables créées dynamiquement.
Tout processus contient également des informations de contrôle réunies dans une structure typiquement appelée PCB (Process Control Block). Cette zone contient l'état du processus, un compteur d'instructions, le contenu des registres

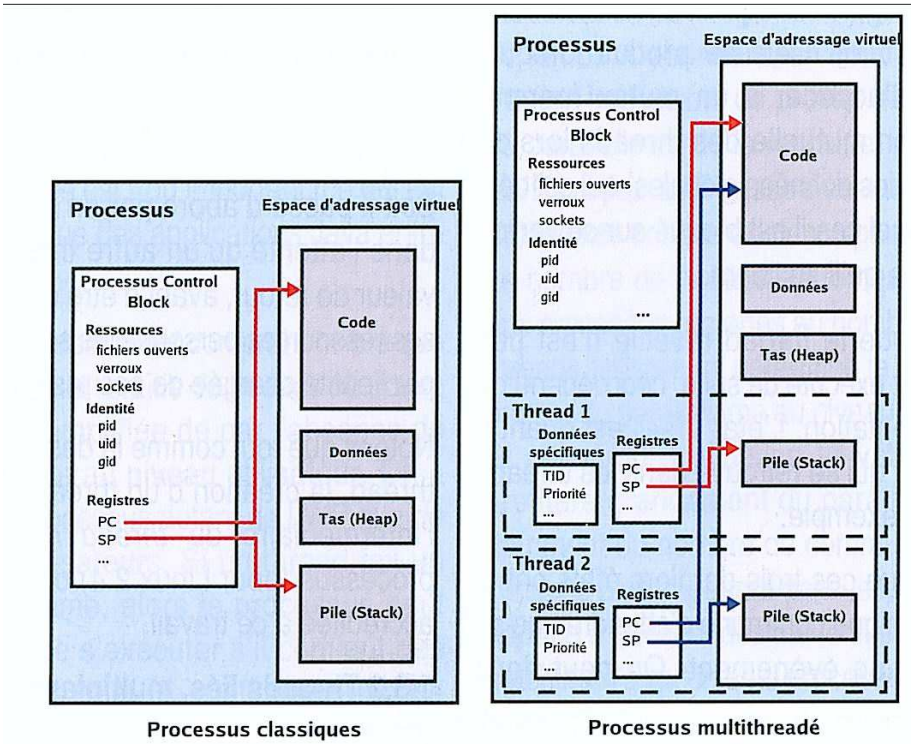


Fig. 12.1 – Processus classique et multithreadé

de l'UC pour pouvoir les restaurer lors de l'ordonnancement. En plus de ces informatons on trouve des informations relatives aux entrées/sorties, à la gestion mémoire, à l'ordonnancement et à la comptabilistion des ressources conssomées.

Dasn un processus multithreadé, plusieurs flots d'exécution se partagent les mêmes ressources (descripteurs de fichiers, espaces d'adressage des variables...). Cependant pour permettre l'exécution simultanée de ces flots, chacun possède un identifiant unique, une pile d'exécution propore, des registres (pointeur de pile...), un état...

Chaque thread possède également des données privées (dans la norme POSIX).

On peut résumer en disant qu'un thread est une entité d'exécution , rattachée à un processus, chargée d'exécuter une partie du processus. Ce dernier étatn vu comme un ensemble de ressources (espace d'adressage, fichiers, périphériques...) que ses threads se partagent.

12.3.2 Les threads en théorie : threads versus processus

Les threads apportent un certain nombres d'avantages quand il s'agit de developper des applications parallèles. Il est plus aisé etplus rapide de créer ou de détruire des threads. Sur une machine SMP (Symetric Multi Processor), plusieursthreads peivent s'exécuter en simultané sur différents processeurs.

Les parties indépendantes des applications peuvent être avantageusement implémentées dasn des threads différents.

Quand un thread est bloqué, par exemple à cause d'une attente d'E/S, l'exécution peut être rtansférée à un autre thread de la même application (au lieu de passer à un autre processus), permetant de profiter pleinement du *timeslice* (tranche de temps) accordé au processus par le *scheduler* (ordonnanceur).

De plus les threads d'un même processus partageant le même espace d'adressage peuvent communiquer entre eux sans faire appel au kernel (gain de temps par rapport à la communication entre processus).

Ce dernier point fait apparaitre la nécessité de synchroniser l'accès aux ressources pour éviter la corruption des données. Il est alors nécesaires d'utiliser des verrous. Il est à noter que ces verrours sont d'autant plus efficcace qu'il ne nécessitent pas l'intervention du kernel.

12.3.3 Les threads en théorie : implémentation des threads

La prise en charge des threads dans le système peut varier selon l'implémentation retenue par les concepteurs . La conformité avec POSIX peut être obtenue sans qu'il en dépende de l'implémentation. Cette norme impose uniquement l'API : le systèmes est donc vu comme une boîte noire par POSIX.

12.3.3.1 Les états d'un thread

Quel que soit le modèle de multithreading, on retrouve toujous les mêmes états principaux pour les threads. Ils permettent, tout comme ils le font pour les processus, de renseigner l'ordonnanceur pour qu'il sache quelle tâche lancer.

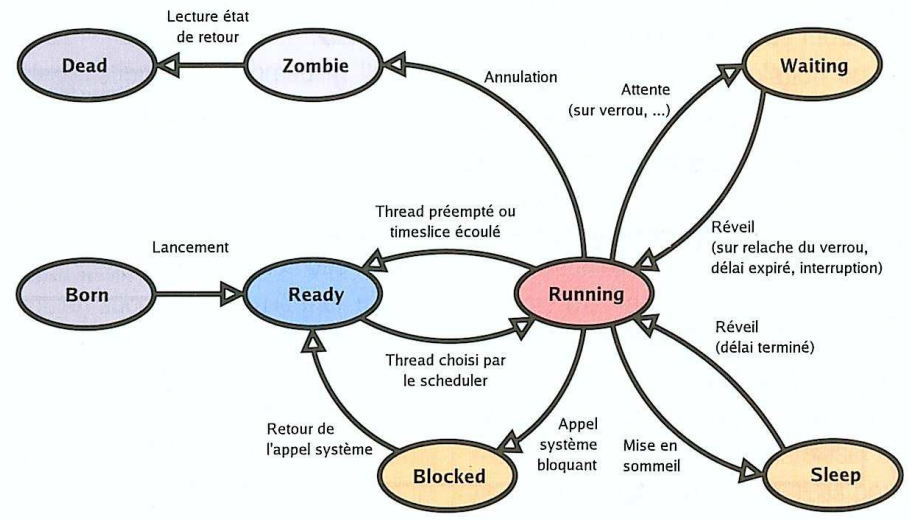


FIG. 12.2 – État d'un thread

On retrouve pour chaque thread trois états clés : `running`, `ready` et `blocked`. L'état `suspend` que peut avoir un processus n'existe pas (l'état `waiting` est une suspension mais n'a pas la même sémantique) ; en effet suspendre un unique thread revient à suspendre l'exécution cd tous les threads d'un même processus, puisque celui-ci se retrouve *swappé* pour laisser la place à d'autres processusen mémoire vive.
L'état `running`est attribué à un thread en cours d'exécution, c'est à dire quand il s'exécute réellemet sur le processeur. Il faut tout de mêm faire attention, car il reta possible que l'état d'un thread soit différent de celui de son processus suivant l'implémentation multithreading retenue (cas des *green threads*).
Tout thread pouvant être choisi par l'ordonnanceur est dans l'état `ready`.
L'état `blocked` intervient lorsqu'un thread fai tun appel d'E/S, par exemple, ou un quelconque autre appel système, ou que sa rtanche de temps impartie est écoulée. Suivant le modèle utilisé, ce *timeslice* est calculé : soit au niveau de l'ordonnancement au sein des processus, auquel cas il s'agit d'une décision de l'espace utilisateur ; soit dans l'espace du noyau.
Il se peut également qu'un threadsoit dans l'état `waiting`. Cela se produit lorsqu'il essaie d'accéder à un mutex (permet l'exlusion mutuelle des threads lors de l'accès à des données globales) qui est déjà pris auquel il est bloquésur ce verrou jusqu'à ce qu'il se libère.
Notons que le thread réveillé n'est pas forcément exécuté de suite. Ceci dépend de l'implémentation. L'état `sleep` est un état qui se retrouve dans les threads Java par exemple.
Ces trois derniers états ont la caractéristique d'attendre qu'un évènement se produise.
Lorqu'un thread parvient a son terme ou lorsqu'il est tué par un autre thread, deux situations peuvent se produire. Soit ses ressources sont libérées directement par l'entité en charge (en général, il s'agit du kernel, mais dans le cas des noyaux linux 2.4, un *thread manager* associé au processus relatif au thread à supprimer se charge de la libération). Ce cas n'est possible que si le thread possède l'attribut `detached`.
Soit il passe d'abord par un état *zombie*, dans l'attente qu'un autre thread lise sa valeur de retour, avant d'être détruit et que ses ressources personnelles soient libérées par l'entité chargée pour ces responsabilités.
Tout comme la destruction d'un thread, la création se fait par l'intermédiairedu *thread manager* du processus ou par l'entité accréditée à ce travail.

12.3.3.2 Threads liés, multiplexés et green threads

On considère généralement trois modèles de multithreading. Leurs différences sont fonctions de la prise en charge plus ou moins importante du kernel. Les threads ont toujours une existence dans l'espace utilisateur, ils sont reliés (en fonction du modèle) à des threads systèmes qui sont l'entité de base du *scheduler*.
Ce sont ces threads qui vont être multiplexés sur les différents processeurs. La liaison entre ces deux sortes de threads se fait par le biais des LWP (Light-Weight Process).

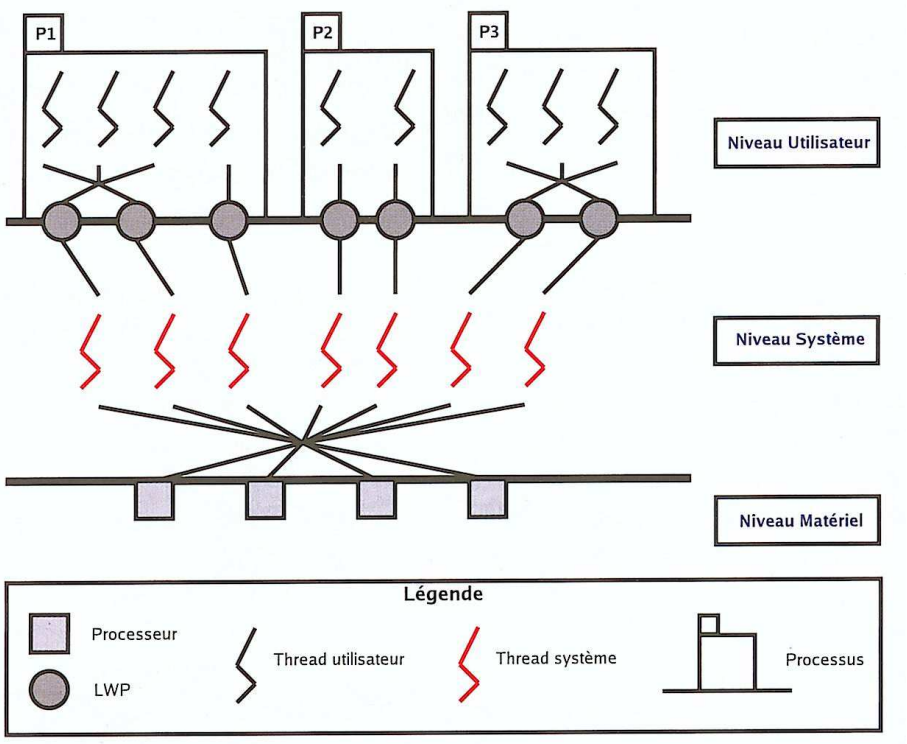


FIG. 12.3 – État d'un thread

Les threads liés Analysons le modèles des threads liés (figure 12.3 P2). C'est l'implémentation la moins complexe et s'avère être un choix judicieux lorsque l'ordonnanceur système est de complexité constante ($O(1)$). Il s'agit d'ailleurs du modèle retenu pour linux dans la série des 2.5.x et 2.6.x, l'ordonnanceur aynat été grandement modifié et etant dorénavant en ($O(1)$).
Il s'agit tout simplement d'assoscier à un thread utilisateur un thread unique système. On qualifie souvent ce modèle de 1-à-1. L'importantce d'voir un ordonnanceur de complexité constante se comprend car le nombre de threads systèmes est exactement égal au nombre de threads utilisateurs ; par conséquent, on évite l'écroulement du systèmes si le temps mis par l'ordonnanceur à faire son choix est indépendant du nombre de threads systèmes. Si ce n'est pas le cas, le systèmes des threads liés mis en place se trouve fortement affecté lorsqu'un nombre de threads créés est trop important (ce qui est caractéristique des applications Java en autres).
Dans ce modèle, il est compréhensible que la gestion de la librairie de threads se trouve fortement simplifiée de par l'absence du multiplexage au niveau utilisateur . On a donc par processus autant de LWP que de threads utilisateurs. Si un thread fait un appel système, alors le processus peut continuer de s'exécuter s'il contient des threads à l'état `ready` et que sa tranche de temps impartie n'est pas écoulée. Le cas échéant , seul le LWP associé au thread ayant effectué l'appel système est alors bloqué, laissant libre cours aux autres LWP de s'exécuter à sa place au sein du même processus. Le blocage dû aus apples systèmes se fait au niveau des threads. Notons également que les routines du kernel peuvent être multithreadées. Ce modèle est adopté par Linux 2.4 et 2.6 et Solaris 8.

Les threads multiplexés Le modèle des threads multiplexés (Figure 12.3 P3) est le plus compliqué à mettre en œuvre, il s'agit du modèle retenu dans Solaris 2 et Windows XP entre autres. L'ordonnancement ici se fait à deux niveaux. En effet, au niveau système, les LWP (associés au thread système) sont multiplexés et on ordonnance aussi les threads utilisateurs sur les LWP. On obtient ici une flexibilité accrue, mais on perd en efficacité. Un autre intérêt vient de la possibilté d'avoir une adaptation de la politique d'ordonancement au niveau utilisateur en fonction de l'application considérée. Les temps dus aux changements de contexte lors de l'ordonnancement des threads sont fortement réduits ; en effet , la majeure partie de la gestion peut se faire dans l'espace utilisateur (s'il n'y a pas d'ajout de LWP au processus courant), le multiplexage des threads sur les différents LWP n'est pas connu du kernel, et donc il n'y a pas de changement de contexte de sa part (pas de modification de registres...), d'où ule gain de temps. Cependant, en contre-partie, il s'avère impossible de profiter entièrement du parallélisme de la machine, car seul le LWP ou les LWP associés aux threads multiplexés ont une existence pour le kernel.
Le nombre de threads systèmes accordés aux processeurs ramenés au nombre total de threads utilisateurs, détermine la capacité à profiter du parallélisme au niveau *hardware*. Si ce nombre avoisine la valeur 1, on profitera grandement du parallélisme au détriment du nombre de commutations de contexte plus important dans le kernel. Tout dépend à quoi est destiné le système. Si la machine n'a qu'un seul processeur, alors il paraît avantageux d'opter pour peu de

LWP par processus.
Dans Solaris 2 (entre autres) le kernel dispose d'un *pool* (bassin) de threads (systèmes) qu'il accorde au processus en fonction des demandes de la librairie. Si tous les LWP sont bloqués et qu'un thread est prêt à l'exécution , un autre LWP est fourni au processus pour exécuter ce thread. Les LWP inutilisés pendant un certain temps sont éliminés. La bibliothèque se charge donc d'ajuster dynamiquement le nombre de LWP fournis au processus.

Les threads liés et multiplexés Il est également possible d'avoir des processus hybrides (figure 12.3 P1) dans ce modèle, c'est à dire contenant à la fois des threads liés et des threads multiplexés.

Les Users Level Threads ou Green Threads Le modèle des *Users Level Threads* ou encore *Green Threads* permet une utilisation sur des systèmes ne comportant aucune fonctionnalité favorisant l'implémentation du multithreading. Ils sont implémentés uniquement dans un espace utilisateur, d'ou une portabilité accrue. Ils ne sont pas connus du noyau, leur existence est simulée uniquement dans une bibliothèque. Toutes la gestion des threads ne requiert aucun appel au kernel, d'où un gain de temps. Cependant , il s'agit là d'une maigre compensation, car il est alors impossible d'assigner à différents processeurs différents threads du même processus simultanément. En outre si un threads fait un appel système, il y a de grandes chances que la kernel passe la main à un autre processus, car il n'a pas la connaissance de la segmentation en threads du processus. Par conséquent, il modifiera l'état du processus si celui-ci effectue un appell d'E/S par exemple, et fera appel au *scheduler* pour décider à qui passer la main. Le thread aura quant à lui toujours un état `running`, son état est donc indépendant de l'état du processus. On perd ici la possibilité de profiter pleinement du *timeslice* accordé au processus.
Notons que la commande `ps -L` permet de voir l'ensemble des LWP alloués par le kernel aux différents processus. On constate alors aisément que le nombre de threads sous linux correspond au nombre de LWP : on se trouve bien en présence d'un modèle 1-à-1.

12.3.3.3 Les attributs des threads

Divers attributs permeteent de modifier le comportement des threads, on se limitera ici aux attributs POSIX.
Començons par l'attribut de priorité d'un threads (pas encore supporté par le kernel 2.6.x qui manque de support approprié pour le temps réel), qui se retrouve dans les systèmes temps réel, dans lequels on attribue à des tâches critiques une priorité forte pour qu'elles puissent *préempter* les tâches de criticité moindre.
Il existe aussi des attributs modifiants le comportement de l'annulation ou de la suppression d'un thread. Nous avons vu qu'il était possible de détacher un thread du contexte, dans le sens ou sa terminaison n'entraîne pas un passage à l'état *zombie*, c'est à dire qu'aucun autre thread du même processus ne peut accéder à la valeur de retour du thread détaché. Ceci peut être intéressant lorsque des tâches n'ont pas besoin de se synchroniser à leur fin. La libération des ressources peut se faire dès la terminaison d'un thread détaché. Les deux attributs complémentaires associés sont `PTHREAD_CREATE_DETACHED` et `PTHREAD_CREATE_JOINABLE`.
Un thread peut clairement annoncer son refus d'être annulé à ses dépends (attribut `PTHREAD_CANCEL_DISABLE`) ou accepter toute demande d'annulation (`PTHREAD_CANCEL_ENABLE`). Le cas échéant, il reste possible de différer (`PTHREAD_CANCEL_DEFFERED`) cette annulation jusqu'à un point d'annulation (4 fonctions POSIX et certains appels systèmes qui peuvent être bloquant comme `read()`), évitant ainsi qu'un thread se voit tué dans une situation critique (accès en écriture à une variableglobale par exemple). Si les fonctions mises en jeu dans le corps du thread le permettent, on pourra opter pour un comportement asynchrone (`PTHREAD_CANCEL_ASYNCHRONOUS`) lors de l'annulation d'un thread, autrement dit d'une annulation immédiate.
Les autres attributs des threads correspondent à leur ordonnancement. On peut agir tout d'abord sur la politique d'ordonnancement : `SCHED_FIFO` (ordonnancement de type *premier arrivé, premier servi*), `SCHED_RR` (*Round Robin* ou tourniquet, *i.e.* à temps partagé) et `SCHED_OTHER` (varie selon la bibliothèque de thre ads ou le noyau utilisé).
L'ordonnancement peut être ensuite effectué sur l'ensemble des threads des processus, auquel cas il a une portée système (`PTHREAD_SCOPE_SYSTEM`), ou au sein du processus (`PTHREAD_SCOPE_PROCESS`). Ceci dépend fortement de la bibliothèque de multithreading utilisée : par exemple LinuxThreads comme NTPL(Native POSIX Thread Library) ne supporte pas `PTHREAD_SCOPE_PROCESS`. En effet, il n'existe pas d'ordonnancement au niveau utilisateur. L'attribut sur la portée n'a donc de sens que sur des systèmes où le multithreading est hybride. Le cas échéant, les priorités des threads sont considérés uniquement au sein d'un processus ou au regard des priorités de tous les threads du système.
Finalement on peut choisir d'hériter de la politique d'ordonnancement du threads père (`PTHREAD_INHERIT_SCHED`) à n'importe quel moment de l'exécution et de revenir ensuite sur sa propre configuration (`PTHREAD_EXPLICIT_SCHED`).
Notons également la possibilité de modifier la taille maximale de la pile associée à un thread via l'attribut `stacksize`.

12.3.4 Le cas de Linux 2.4

Dans cette version, Linux ne propose pas de système de multithreading efficace. Des lacunes dans le noyau entrainaient une limitation pour l'écriture d'une bibliothèque compatible POSIX. L'appel système `clone()` servant à la création de nouveaux processus a depuis été modifiée facilitant l'implémantatin du multithreading.

Le système mis en place est un modèle 1-à-1, autrement dit les processus sont composés de threads liés uniquement. De plus, le manque de support de la part du noyau oblige à l'utilisation d'un *thread manager* au sein de chaque processus. Il a pour rôle de créer et de détruire les nouveaux threads, assure aussi l'implémentation correcte de la sémantique des signaux, ainsi que d'autres parties de gestion. Mais il ajoute une lourdeur au système, et ralentit le processus de création et de destruction.
L'absence de moyen pour la synchronisation au sein du noyau amène à l'utilisation des signaux dans l'implémentation des threads.
Ce pendant la gestion des signaux au sein de la bibliothèque de threads est fragile et non conforme POSIX. Ceci est dû à l'absence de concept de groupes de threads dans le noyau.
L'ABI (Application Binary Interface) de ELF (le format binaire couramment utilisé sous linux), n'est pas prévue pour le stockage des données privées aux threads.
Pour remédier à cela, des relations fixes entre le pointeur de pile et la position du descripteur de thread sont utilisés. A ceci s'ajoute un nombre limité de threads.
Concluons sur les problèmes de cette implémentation :
- Le *thread manager* est un goulot d'étranglement pour la création et la destruction de threads ; de plus le nettoyage du processus appartient à ce thread qui, s'il se trouve tué, ne peut plus agir.
- On ne peut pas envoyer un signal à un processuscomme un tout (chaque thread à un PID différent).
- L'utilisation des signaux comme primitives de synchronisation aboutit à de gros problèmes. Des réveil hasardeux peuvent se produire, ce qui engendre une pression supplémentaire sur le système de gestion des signaux du kernel.
- `SIGSTOP` et `SIGCONT` stoppent uniquement un thread et non l'ensemble du processus.
- Chaque thread possède un PID différent (problème avec les signaux). Une limite de (8192-1) threads sur l'IA32 (*Intel Architecture*. Le système de fichier `/proc` devient dur à utiliser, si trop de threads sont présents.
- Le manque d support du noyau empêche l'implémentation correcte des signaux. En outre, leur utilisation est une approche lourde.

12.3.5 La librairie NTPL (Ingo Molinar et Ulrich Drepper) et les modifications apportées au noyau Linux

Le but de cette nouvelle bibliothèque est d'assurer une compatibilité POSIX, une utilisation efficace des systèmes multiprocesseurs (SMP), des coûts de création de threads faibles, une compatibilité binaire maximale avec l'implémentation Linux Threads, un coût administratif non proportionnel au nombre de processeurs utilisés, le support NUMA (*Non-Uniform Memory Architecture*), etc...
Notons également l'intégration dans C++ pour la gestion eds exceptions où le nettoyage des objets s'apparente à la destruction de threads.

12.3.5.1 Les nouveautés de la NTPL (Ulrich Drepper, 2003)

Le modèle de threads choisis est du 1-à-1. La fléxibilité d'un modèle plusieur-à-plusieurs (hybride ou multiplexé) vient avec une complexité de l'implémentation du système.
De plus l'amélioration de l'ordonnanceur (complexité en temps constante) permet de rendre tout à fait efficace un système basé uniquement sur des threads liés. La gestion des signaux avec un système multiplexé aurait pu être réalisé au niveau utilisateur, cependant on aboutirait à une complexité et des retards dans la gestion des signaux.
NTPL en finit donc avec la limitation du nombre de threads, et la gestion des signaux problématiques.
Cette dernière tâche incombe au noyau, qui doit gérer la multitude de masques de signaux. Il s'ensuit quand même la suppression des retards dans la délivrance des signaux (un isgnal n'est émis qu'à un thread non bloquant).
NTPL supprime l'existence du *thread manager* (le noyau assure maintenant la désallocation des ressources des threads, ainsi qu'un support correct des signaux POSIX).
Un autre problème du *thread manager* est qu'il ralentit la création des threads. En effet, toute requête de création est traitée par lui de manière obligatoirement séquentielle, car il ne peut s'exécuter que sur un seul processeur. De plus,trop de responsabilités lui sont données, ce qui augmente de manière significative la durée d'un changement de contexte.
Les primitives de synchronisation sont désormais implémentées à l'aide de FUTEX (*Fasr User muTEX*), qui est une nouvelles fonctionnalité ajoutée au noyau (au cours de la série 2.5.x). Le mécanisme, bien que simple, permet l'implémentation de toutes ces primitives. Les threads appelants bloqués peuvent être réveillés à l'issue d'une interruption ou après un délai. En outre, la gestion de cette synchronisation peut se faire presque dans sont intégralité au niveau utilisateur, d'où un gain de rapidité. Les Futex sont faits pour fonctionner en mémoire partagée, ce qui permet la communication POSIX inter-processus.
Le problème de rendre la création de threads rapide est résolu en stockant les structures de données des threads et leurs données privées sur la pile (une modification de l'ABI de ELF est nécessaire), ainsi qu'en évitatn de libérer directement la mémoire lorsqu'un thread est tué, en vue d'une réutilisation.

12.3.5.2 Les modifications relatives dans le noyau Linux

Elles furent menées pour la plupart par Ingo Molnar, en vue d'assurer une interface optimale entre la bibliothèque et le kernel. Voici les modifications effectuées :
- Le support d'un nombre arbitraire de zones de données spécifiques à chaque thread pour l'IA32 et X86-64.
- L'appel système `clone` est étendu pour optimiser la création de nouveaux threads et leur destruction sans l'aide d'un autre thread. Le kernel stocke le TID (*Thread IDentifier*) d'un nouveau thread à une adresse mémoire donnée et efface son contenu lorsque le thread est tué. Ceci aide à l'implémentation d'une gestion de mémoire en espace utilisateur sans intervantion du kernel.
- La gestion des signaux POSIX pour les processus multithreadés est supportée par le noyau. Les signaux sont maintenant délivrés à un thread actif du processus, un signal fatal détruit l'intégralité du processus. Les signaux `SIGSTOP` et `SIGCONT` affectent maintenant entièrement le processus.
- L'appel système `exit()` termine désormais le thread appelant, et le nouvel appel introduit `exit_group()`, finit le processus. De plus, le temps mis par cet appel pour stopper un processus avec beaucoup de threads a été grandement réduit.
- L'appel système `exec()` utilise maintenant le même PID que le processus original, et tous les threads présents sont terminés avant que la nouvelle image du pocessus acquière le contrôle.
- Les statistiques sur l'usage des ressources affectent entièrement le processus.
- L'implémentation de `/proc` a étét améliorée pour faire face à de possibles milliers d'entrées générées. Chaque thread a un sous répertoire dans celui de son processus.
- Sur un `exit()`, les threads détachés sont supprimés via un reveil sur Futex par le noyau, ce qui correspond à un «join» du noyau (avant utilisation d'un *thread manager*).
- L'espace des PID a étét étendu à un maximum de 2 milliards sur l'IA32 et `/proc` peut contenir jusqu'à 64000 processus.

12.3.5.3 Ce qu'il manque pour un «100% POSIX compliant»

Malgré toutes ces améliorations, la compatibilité POSIX n'est pas totalement assurée. En effet quelques défis restent encore à relever. Voici la liste des lacunes :
- La famille des appels systèmes `setiud()` et `setgid()` doivent affecter le processus entier et non seulement le threads appelant.
- Le niveau `nice` est une composante globale du processus.
- La limite d'utilisation du CPU doit être une limite sur le temps que dépensent tous les threads d'un processus ensemble.
- Le manque de support du noyau pour le temps réel empêche NTPL de supporter certaines fonctionnalités. Bien que les appels systèmes pour modifier le scheduling soient présents, ils n'ont pas d'effet garanti car la majorité du kernel ne suit pas les règle de l'ordonnancement temps réel. Par exemple, reveiller un thread sur un Futex se fait sans regard des priorités.

12.4 L'API des *pthreads* (*POSIX threads*)

Les *pthreads* sont la normalisation IEEE des threads. Cette norme précise le comportement en boîte noire, autrement dit l'implémentation des fonctions peut différer d'une bibliothèque compatibles POSIX à une autre.
En ce qui concerne Linux, la gestion des signaux n'est correctement implémentée que depuis NTPL. Les priorités sur les *mutex* et *variables de conditions* ne sont pas effectives.
En effet, Linux manque encore de support pour le temps réel, par conséquent NTPL ne peut pas supporter ce qui n'est pas pris en charge par le noyau.
Notons toutefois que le partage de ces entités de synchronisation entre différents processus est géré depuis la NTPL de par l'émergence des Futex dans le noyau.

12.4.1 Opérations classiques et avancées

Pour la compilation d'un programme multithreadé, ajouter sur la ligne du compilateur `-D_REENTRANT` (cela permet un comportment correct de certaines macros), et sur la ligne de commande de l'éditeur de lien ajouter `-lpthread`.
Notons qu'un code est dit réentrant s'il peut être exécuté de manière simultanée par plusieurs tâches sans que ceci engendre des corruptions de données, de blocage, ou tout autre effet non désirable.
Toutes les primitives suivantes sont déclarées dans le fichier d'entête `pthread.h`

12.4.1.1 Gestion des threads

```
int  pthread_create(pthread_t  *  thread, pthread_attr_t * attr, void *
```

```
(*start_routine)(void *), void * arg);
```

Le TID est renvoyé dans `thread`, on peut également spécifier une structure d'attributs (ou `NULL`) et la routine à exécuter peut disposer d'un argument.

```
void pthread_exit(void *retval);
```

Termine le thread appelant et renvoie `retval`.

```
int pthread_join(pthread_t th, void **thread_return);
```

Attend la fin d'un thread et lit sa valeur de retour.

```
int pthread_detach(pthread_t th);
```

Permet de détacher un thread, il ne peut plus communiquer sa valeur de retour à un autre thread.

```
pthread_t pthread_self(void);
```

Retourne le TID du thread appelant.

```
int pthread_equal(pthread_t thread1, pthread_t thread2);
```

Compare deux TID et renvoie une valeur non nulle s'il sont égaux.

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;

int  pthread_once(pthread_once_t  *once_control,  void  (*init_routine) (void));
```

Assure que la fonction passée en paramètre ne s'exécute qu'une seule fois. Cette routine est typiquement utilisée pour des fonctions d'initialisation comme l'allocation d'une structure globale. Parmi tous les threads appelant `pthread_once` avec la même variable de contrôle, un seul l'exécute. On déclare la variable `once_control` de manière statique.

```
int pthread_kill(pthread_t thread, int signo);
```

Envoir le signal `signo` au thread passé en argument.

```
void pthread_yield(void);
```

Le thread appelant libère le processeur sur lequel il s'exécute, il est placer dans la file `ready` (*runnable*) du scheduler qui va choisir un autre thread à exécuter (cette fonction est une extension GNU).

12.4.1.2 Annulation d'un thread

```
int pthread_cancel(pthread_t thread);
```

Emet une demande d'annulation au thread passé en argument.

```
void pthread_cleanup_push(void (*routine) (void *), void *arg);
void pthread_cleanup_pop(int execute);
```

Lorsque le thread reçoit une demande d'annulation, les routines empilées sont exécutées. Ces deux fonctions s'utilisent ensemble : on empile une fonction de nettoyage à l'entrée d'une section et on la dépile lorsqu'on sort. Lors d'un *pop*, on peut soit exécuter la fonction empilée (`execute=1`) soit la dépiler sans l'exécuter (`execute=0`).

```
int pthread_setcancelstate(int state, int *oldstate);
```

Le thread peut soit accepter (`PTHREAD_CANCEL_ENABLE`) ou refuser (`PTHREAD_CANCEL_DISABLE`) les demandes d'annulation de la part d'autres threads.

```
int pthread_setcanceltype(int type, int *oldtype);
```

Si on autorise l'annulation, elle peut soit prendre un effet immédiat (`PTHREAD_CANCEL_ASYNCHRONOUS`), soit lorsque l'exécution arrive à un point d'annulation (`PTHREAD_CANCEL_DEFERRED`) : il s'agit de certaines fonctions systèmes bloquantes, et quatre fonctions des pthreads.

```
void pthread_testcancel(void);
```

Teste si une damande d'annulation a été émise (point d'annulation).

12.4.1.3 Ordonnancement des threads

```
int  pthread_getschedparam(pthread_t target_thread, int *policy, struct
sched_param *param);
int pthread_setschedparam(pthread_t target_thread,  int  policy,  const
struct sched_param *param);
```

La première fonction récupère la politique d'ordonnancement et la seconde l'établit. Trois politiques sont prévues : `SCHED_FIFO` (premier arrivé, premier servi), `SCHED_RR` (*round robin*, *i.e.* temps partagé, `SCHED_OTHER` (politique dependant du constructeur). L'attribut `param` contient principalement la priorité du processus.

12.4.1.4 Réception des signaux

Permet de positionner le masque des signaux que peuvent recevoir le thread appelant (fonctionne comme `sigprocmask()`). Le premier argument renseigne s'il faut ajouter (`SIG_BLOCK`), soustraire (`SIG_UNBLOCK`) au masque courant ou prendre comme tel (`SIG_SETMASK`) le masque passé en second argument.

12.4.1.5 Duplication de processus

```
int  pthread_atfork(void  (*prepare)(void),  void (*parent)(void), void
       (*child)(void));
```

Cette fonction enregistre les trois routines passées en argument. Lors d'un appel `fork()`, avant duplication du processus, la fonction `prepare()` est appelée.
Ensuite `parent()` est exécutée dans le processus père et `child()` est exécutée dans le processus fils.
Cette fonction est utile lorsqu'est mis en jeu un verrou au sein d'un processus entre plusieurs threads, et qu'un threads appelle `fork()`.
Lors de cet appel, un processus fils est créé mais avec seulement un flot d'exécution ; celui du threads appelant .
Si ce thread (dans le contexte du processus fils) veut récupérer le verrou acquis dans le processus père par un autre thread, alors il va bloquer indéfiniment.
Pour remédier à cela `prepare` doit acquérir le verrou (mutex) en question, e,suite les deux autres routines devront libérer ce verrou dansnle contexte du père pour l'une et du fils pour l'autre.

12.4.2 Synchronisation

12.4.2.1 Les mutex : Opérations sur un mutex

```
int  pthread_mutex_init(pthread_mutex_t  *mutex,  const  pthread_mutex-
       attr_t *mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Permet de créer ou de détruire un mutex de manière dynamique. On alloue préallablement de la mémoire via `malloc` avec pour argument `sizeof(pthread_mutex_t)`. La création d'un mutex peit être paramétrée (sinon positionner le 2ème paramètre à `NULL`). Pour initialiser un mutex de manière statique on procède comme suit :

```
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Le première bloque jusqu'à obtenir le mutex, la seconde le relâche, et la troisième tente d'obtenir le mutex et echoue si elle ne le peut.

12.4.2.2 Les mutex : Gestion des mutex

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

Permet la création ou la destruction d'une structure d'atrtibuts à passer lors de la création d'un mutex.

12.4.2.3 Les mutex : Partage d'un mutex

```
int pthread_mutexattr_getpshared(const pthread_mutexattr_t *attr, int * pshared);
int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);
```

Le partage d'un mutex ebtre plusieurs processus se fait par l'intermédiairede ces fonctions. L'argument `pshared` peut prendre la valeur `PTHREAD_PROCESS_SHARED` pour en activer l apartage ou la valeur `PTHREAD_PROCESS_PRIVATE` pour en restreindre l'accès au processus l'ayant créé. Notons que la création d'un mutex doit se faire en mémoire partagée si l'on souhaite activer l'option `PTHREAD_PROCESS_SHARED`.
Attention, sous linux cette fonctionnalité n'est disponible que depuis NTPL.

12.4.2.4 Les mutex : Protocoles pour la résolution des priorités

Les fonctions suivante permettent de récupérer ou d'établir le protocole utilisé pour règler les problèmes d'inversion de priorités (*inversion safe*) pouvant survenir lorsqu'un thread de priorité supérieure à un second essaie de prendre un mutex que ce dernier tient.
Dans cce cas, un thread de priorité intermédiaire peut préampter le thread de priorité basse et retarder finalement la prise du mutex par le thread de priorité haute qui est en attente sur le mutex. On risque donc un retard pour l'exécution d'une tâche critique.
Pour résoudre ce problème, un protocole simple donne temporairement une priorité au thread ayant le mutex égale à celui du thread de priorité haute (mécanisme d'héritage de la priorité). On peut également inverser les priorités des deux tâches de manière à l'aisser s'exécuter le thread de priorité «initialement» basse jusqu'à ce qu'il relâche le mutex (mécanisme d'inversion des priorités). On résout ainsi le problème du thread de priorité intermédiaire. Il s'agit ici de considération temps réel, aussi ces fonctions ne sont implémentées que pour des systèmes de ce type.

```
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr, int * protocol);
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, int protocol);
```

Les trois valeurs de `protocol` sont : `PTHREAD_PRIO_NONE` (pour aucun protocole), `PTREAD_PRIO_INHERIT` (protection par héritage des priorités) et `PTHREAD_PRIO_PROTECT` (protection par inversion des priorités).

12.4.2.5 Les mutex : Configuration de la priorité

```
int pthread_mutexattr_getprotocolprioceiling(pthread_mutexattr_t *attr,
int prioceiling, int * oldceiling);
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr, int prioceiling);
```

Le valeur `prioceiling` (de 1 à 127) doit correspondre à la priorité maximale des threads pouvant obtenir le mutex. Cette valeur est ensuite utilisée par le protocole de protection d'inversion pour empêcher les problèmes.

12.4.2.6 Les variables de conditions : Gestion classique

```
int pthread_cond_init(pthread_cond_t    *cond,    pthread_condattr_t
       *cond_attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Permet de créer ou dsétruire une *variable de condition*. En outre, sa création peut être paramétrée (sinon positionner le 2ème argument à `NULL`). Pour initialiser une *variable de condition* de manière statique :

```
 pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

Ces deux fonctions permettent la synchronisation de plusieurs threads. Détaillons leur comportement.
La première débloque le mutex passé en paramètre (il faut que le thread appelant ait initialement le mutex), puis se met en attente sur la *variable de condition*. C'est alors qu'un aure thread peut acquérir le mutex puis effectuer des modifications sur une structure d'échange par exmeple, et finalement appelle `pthread_cond_signal()`.
Le premier thread se réveillle et essaie d'obtenir le mutex (le réveil et le blocage sur le mutex se fait atomiquement dans la fonction `pthread_cond_wait()`), en vain... Il faut d'abord que l'autre thread le relâche. Notons que si plusieurs threads sont en attente sur une même *variable de condition*, alors la fonction `pthread_cond_signal()` n'en réveillera qu'un . Pour réveiller tous les threads en attente, on utilise la fonction suivante :

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Une alternative à la fonction d'attente est sa version temporisée. La fonction suivante attend jusqu'à la date précisée en troisième argument. Pour obtenir la date actuelle on peut utiliser `gettimeofday()`.

```
int   pthread_cond_timedwait(pthread_cond_t   *cond,    pthread_mutex_t
       *mutex, const struct timespec *abstime);
```

Notons qu'une demande d'annulation d'un thread en attente sur une *variable de condition* peut bloque indéfiniment. Il faut d'abord que la fonction `pthread_cond_wait()` ait récupérer le mutex (état prévisible). Cependant, il ne faut pas que le thread se termine avec le mutex bloqué. On utilise donc une fonction de nettoyage comme suit :

```
pthread_mutex_lock(&mutex);
pthread_cancel_push(pthread_mutex_unlock, (void *) &mutex);
while(condition_non_realisee)
pthread_cond_wait(&cond,&mutex);
pthread_cancel_pop(1);
```

12.4.2.7 Les variables de conditions : Utilisation d'attributs

```
int pthread_condattr_init(pthread_condattr_t *attr);
int pthread_condattr_destroy(pthread_condattr_t *attr);
```

Permet la création ou la destruction d'une structure d'attributs à passer lors de la création d'une *variable de condition*.

12.4.2.8 Les variables de conditions : Le partage d'une variable de condition

```
int pthread_condattr_getpshared(const pthread_condattr_t *attr, int *pshared);
int pthread_condattr_setpshared(const pthread_condattr_t *attr, int pshared);
```

Ces fonctions permettent respectivment de récupérer et de configurer l'état de partage d'une *variable de condition*. L'argument `pshared` peut prendre la valeur `PTREAD_PROCESS_SHARED` pour en activer la partage ou la valeur `PTREAD_PROCESS_PRIVATE` pour en restreindre l'accès au processus l'ayant créé.

12.4.3 Les données privées

Notons que l'utilisation dans la déclaration des variables du mot clé `__thread` permet de déclarer statiquement des données privées pour un thread C et C++. Attention tout de même, car cci n'est pas une extension officielle des langages, mais les compilateurs doivent implémenter ceci pour supporter la nouvelle ABI de l'ELF.

```
int pthread_key_create(pthread_key_t *key, void (*destr_function) (void *));
int pthread_key_delete(pthread_key_t key);
```

Permet la création et la destruction de clés pour le stockage de données spécifiques à un thread (analogue à une table d ehachage).
On peut éventuellement spécifier une fonction pour libérer la clé (lors du `delete`) en deuxième argument de la première fonction (sinon `NULL`).

```
void * pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *pointer);
```

Ces fonctionds permettent respactivement de récupérer par une variable privée (via un pointeur), ou de mémoriser la variables passée en paramètre dans une zone privée du thread appelant.

12.4.4 La modification des attributs

12.4.4.1 Gestion de base

```
int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

Permet la création ou la destruction d'une structure d'attributs à passer lors de la création d'un thread.

12.4.4.2 Attributs detached/joinable

```
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
int   pthread_attr_getdetachstate(const   pthread_attr_t   *attr,   int *detachstate);
```

Permet de spécifier si un thread doit être détaché (`PTHREAD_CREATE_DETACHED`) ou *joinable* (`PTHREAD_CREATE_JOINABLE`), ou sinon de connaître cet état.

12.4.4.3 Modification de la pile d'un thread

```
int pthread_attr_getstackaddr(const pthread_attr_t *attr, void **stackaddr);
int pthread_attr_getstacksize(const pthread_attr_t *attr, void *stacksize);
int pthread_attr_setstackaddr(const pthread_attr_t *attr, void *stackaddr);
int pthread_attr_setstacksize(const pthread_attr_t *attr, void stacksize);
```

On récupère ou on modifie soit la taille de la pile pour le thread que l'on souhaite créer, soit l'adresse de la base de la pile.

12.4.4.4 Attribut sur l'ordonnancement

```
int   pthread_attr_setschedparam(pthread_attr_t   *attr,  const  struct
       sched_param *param);
int  pthread_attr_getschedparam(const  pthread_attr_t   *attr,   struct
       sched_param *param);
```

L'attribut modifié ou récupéré concerne esssentiellement la priorité du thread. Le champ associé à la structure est `sched_priority`, sa valeur peut aller de 1 (thread le moins favorisé) à 127 (thread le plus favorisé).

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(const pthread_attr_t *attr,  int  *policy);
```

On choisit ou on récupère ici la politique d'ordonnancement. Trois politique sont prévues : `SCHED_FIFO` (premier arrivé, premier servi), `SCHED_RR` (*round robin*, *i.e.* temps partagé), `SCHED_OTHER` (politique dépendant du constructeur).
On signale (ou on récupère) que l'ordonnancement est spécifique au thread créé (`PTHREAD_EXPLICIT_SCHED`), ou s'il doit suivre la configuration du thread créateur (`PTHREAD_INHERIT_SCHED`).

```
int pthread_attr_setscope(pthread_attr_t *attr, int scope);
int pthread_attr_getscope(const pthread_attr_t *attr, int *scope);
```

Ces fonctions n'ont d'intérêt que pour un multithreading hybride. Le cas échéant, l'ordonnancement peut se faire au niveau de chaque processus (`PTHREAD_SCOPE_PROCESS`), ou au niveau système (`PTHREAD_SCOPE_SYSTEM`) : la portée du scheduler est donc globale.

12.5 Applications

Notons les point forts d'un programme multithreadé. En effet il permet :
- Sa décomposition en tâches n'ayant pas un rapport direct de causalité. La décomposition du programme en éléments distincts facilite sa compréhension (au détriment d'une gestion plus compliquée), sa maintenabilité, sa flexibilité.
- Sa pleine utilisation du parallélisme d'une machine, ou à défaut du profit de la tranche totale de temps alloué au processus par le kernel (utilistion au mieux de la CPU).
- Le partage de la mémoire et des fichiers ouverts du processus entre ses threads permet leur communication sans recours au kernel. Les conséquences de ces rapports nous amènent trivialement à l'utilisation du multithreading dans les serveurs.

Donnons un exemplede scénario possible. Un thread R s'occupe de la reception des requêtes émises par les clients. A chacune d'entre elles un nouveau thread est créé pour la traiter, et peut même être dédié à une communication s'il y a lieu avec le client. De cette manière, le thread R peut s'occuper de nouvelles demandes.
La connexion au serveur se fait toujours par le même canal de communcation (*port*). Les clients dialoguent ensuite avec leurs threads dédiés par d'autres ports, lesquels sont discutés par les clients et le serveur avant le traitement de leur requête.
Il est aussi possible de créer un *pool* (bassin, réserve) de threads, permetttant d'anticiper sur la demande et ainsi de traiter plus rapidement les requêtes des clients. Un autre domaine dans lequel les threads sont courants est celui des interfaces graphiques. Le dialogue avec l'utilisateur se fait par un GUI (graphique user interface). Le programme associe aux actions de l'utilisateur (clic sur bouton, par exemple) une fonction appelée *callback* (fonction reflexe). Il s'agit en fait d'un thread qui est créé lorsque l'évènement à lieu. De cette manière, l'utilisateur peut continuer à interagir avec l'application (l'interface reste active, elle n'est pas gelée par l'exécution d'une requête). Le traitement des demandes faites par l'utilisateur est donc encore une fois détaché de sa reception. Les bibliothèques graphiques GTK+ et Qt utilisent ce principe.
Le multithreadiong est donc à envisager dans tous les systèmes concurrentiels, dès lors qu'il y a partage de ressources.

12.6 Les bibliothèques de *threads*

De nombreux systèmes d'exploitation permettent aujourd'hui la programmation par *threads* : Solaris 5.x de SUN, Windows95/98/NT/XP et bien d'autres (dont LINUX). Dans le cas de Solaris, la bibliothèque de *threads* disponible est conforme à la norme POSIX 1003.1c ce qui assure une certaine portabilité de l'applicatif en cas de portage vers un autre système. Dans le cas des systèmes Microsoft, la bibliothèque utilisée est bien entendu non conforme à cette norme POSIX ! Il existe aujourd'hui diverses bibliothèques permettant de manipuler des *threads* sous LINUX. On dénombre deux principaux types d'implémentations de *threads* :
– Au niveau utilisateur (*user-level*). A ce moment la, la gestion des *threads* est entièrement faite dans l'espace utilisateur.
– Au niveau noyau (*kernel-level*). Dans ce cas, les *threads* sont directement gérés par le noyau.
Dans ce dernier cas, la base de l'implémentation est entre-autres l'appel système clone, également utilisé pour la création de processus classiques :

```
NAME
       clone - create a child process

SYNOPSIS
       #include <linux/sched.h>
       #include <linux/unistd.h>

       pid_t clone(void *sp, unsigned long flags)

DESCRIPTION
       clone  is  an  alternate  interface  to  fork,  with  more
       options.  fork is equivalent to clone(0, SIGCLD|COPYVM).
```

La bibliothèque LinuxThreads développée par Xavier Leroy (`Xavier.Leroy@inria.fr`) est une excellente implémentation de la norme POSIX 1003.1c. Cette bibliothèque est basée sur l'appel système clone. Je ne saurais trop vous conseiller d'utiliser ce produit, ce que nous ferons dans la suite des exemples présentés dans cet article.

12.7 Comment créer des *threads* sous LINUX ?

Voici un petit exemple de programme utilisant deux *threads* d'affichage : **thread1.c**

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

void *my_thread_process (void * arg)
{
  int i;

  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
  }
  pthread_exit (0);
}

main (int ac, char **av)
{
  pthread_t th1, th2;
  void *ret;

  if (pthread_create (&th1, NULL, my_thread_process, "1") < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }

  if (pthread_create (&th2, NULL, my_thread_process, "2") < 0) {
```

```
    fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
  }

  (void)pthread_join (th1, &ret);
  (void)pthread_join (th2, &ret);
}
```

La fonction `pthread_create` permet de créer le *thread* et de l'associer à la fonction `my_thread_process`. On notera que le paramètre `void *arg` est passé au *thread* lors de sa création. Après création des deux *threads*, le programme principal attend la fin des *threads* en utilisant la fonction `pthread_join`.
Taper ou récupérer le programme `thread1.c`.
Pour le compiler ,utiliser la commande `gcc`, par exemple (premier exercice) : `gcc -D_REENTRANT -o thread1 thread1.c -lpthread`.

Que se passe-t-il ? Pourquoi ?
Réponse :

Maintenant, passer la variable `i` en variable globales. Que se passe-t-il ? Pourquoi ?
Réponse :

12.8 Partages des données et synchronisation

12.8.1 Les MUTEX

Le partage de données nécessite parfois (même souvent) d'utiliser des techniques permettant de protéger à un instant donné une variable partagée par plusieurs *threads*. Imaginons un simple tableau d'entier rempli par un *thread* (lent) et lu par un autre (plus rapide). Le *thread* de lecture doit attendre la fin du remplissage du tableau avant d'afficher sont contenu. Pour cela, on peut utiliser le système des MUTEX (MUTual EXclusion) afin de protéger le tableau pendant le temps de son remplissage.
Taper ou récupérer le programme **thread2.c**

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

```
static pthread_mutex_t my_mutex;
static int tab[5];

void *read_tab_process (void * arg)
{
  int i;

  pthread_mutex_lock (&my_mutex);
  for (i = 0 ; i != 5 ; i++)
    printf ("read_process, tab[%d] vaut %d\n", i, tab[i]);
  pthread_mutex_unlock (&my_mutex);
  pthread_exit (0);
}

void *write_tab_process (void * arg)
{
  int i;

  pthread_mutex_lock (&my_mutex);
  for (i = 0 ; i != 5 ; i++) {
    tab[i] = 2 * i;
    printf ("write_process, tab[%d] vaut %d\n", i, tab[i]);
    sleep (1); /* Relentit le thread d'ecriture... */
  }
  pthread_mutex_unlock (&my_mutex);
  pthread_exit (0);
}

main (int ac, char **av)
{
  pthread_t th1, th2;
  void *ret;

  pthread_mutex_init (&my_mutex, NULL);

  if (pthread_create (&th1, NULL, write_tab_process, NULL) < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }

  if (pthread_create (&th2, NULL, read_tab_process, NULL) < 0) {
    fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
  }

  (void)pthread_join (th1, &ret);
  (void)pthread_join (th2, &ret);
}
```

Compiler le programme.
Que se passe-t-il ? Pourquoi ?

Réponse :

Maintenant, enlever les verrouillages dans les 2 *threads*. Recompiler le programme.
Que se passe-t-il ? Pourquoi ?

12.8.2 Les variables de condition

Une condition (variable de condition) est un mécanisme de synchronisation qui permet aux threads de suspendre leur exécution et de relacâcher les processeurs jusqu'à ce qu'un prédicat sur une donée partagée soit satisfait. Les opérations de base sont : signal de la condition (quand le prédicat est devient vrai), attente de la condition qui suspend l'exécution du thread jusqu'a ce qu'un autre thread signale la condition (la rende vrai).
Une variable de condition doit toujours être associée avec un mutex, pour éviter la «condition de course», ou un thread se prépare a attendre une variable de condition et un autre thread signale la condition juste avant le premier thread qui attend activement cette condition.
`pthread_cond_init` initialise cette variable de condition en utilisant les atrtibuts spécifiés dans `cond_attr`, ou les attributs par défaut si `cond_attr` vaut `NULL`. L'implémentation LinuxThreads ne supporte pas les attributs de conditions et le paramètre `cond_attr` est ignoré.
Les variables de type `pthread_cond_t` peuvent être initialisée staiquement en utilisant la constante `PTHREAD_COND_INITIALIZER`.
`pthread_cond_signal` redémarre un des threads qui attendait le variable de condition `cond`. Si aucun thread n'attend la variable de condition, rien ne se passe. Si plusieurs threads attendent cette variable, seulement un seul est redémarré, mais on ne sais pas lequel. `pthread_cond_broadcast` redémarre tous les threads qui attendent la variable de condition.
`pthread_cond_wait` déverouille atomiquement le mutex (come `pthread_unlock_mutex`) et attends que la variable de condition `cond` soit signalée. L'exécution du thread est suspendue et ne consomme alors aucune CPU jusqu'à ce que la variable soit signalée. Le mutex doit être vérouillé par le thread appelant à l'entrée de `pthread_cond_wait`. Avant de retourner dans le thread appelant , `pthread_cond_wait` refait l'acquisition du mutex (comme dans`pthread_lock_mutex`).
Le déverouillage du mutex et l'attente sur la variable de condition sont fait atomiquement. Donc si tous les threads font toujours l'acquisition du mutex avant de signaler le variable de condition, ceci garanti que la condition ne peut pas être signalée (et donc ignorée) entre le moment ou un thread verouille le mutex et le moment ou il attends la variable de condition.
`pthread_cond_timedwait` deverouille le mutex et attend la condition atomiquement comme `pthread_cond_wait`, mais cette fonction borne la durée de l'attente. Si la condition n'a pas été signalée pendant cet intervalle de temps spécifié par `abstime`, le mutex mutex est de nouveau acquis et `pthread_cond_timedwait` retourne l'erreur `ETIMEDOUT`.
Le paramètre `abstime` spécifie le temps absolu, avec la même origine que les fonctions `time(2)` et `gettimeofday(2)` :

un temps absolu de 0 correspond à 00 :00 :00 GMT, January 1, 1970.
`pthread_cond_destroy` détruit une variable de condition, libère les ressources. Aucin thread ne doit attendre la variable qui va être détruite par la fonction`pthread_cond_destroy`. Dans l'implémentation LinuxThreads, aucune ressource n'est associée à la variable de condition, donc pour l'instant la fonction ne fait que vérifier qu'il n'y a aucun thread qui attende la variable.

```
int pthread_cond_init (pthread_cond_t *cond, const pthread_cond_attr *attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Crée une variable condition (on ignorera l'attribut).

```
int pthread_cond_destroy (pthread_cond_t *cond);
```

Détruit la variable condition.

```
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
```

L'activité appelante doit posséder le verrou mutex. L'activité est alors bloquée sur la variable condition après avoir libéré le verrou. L'activité reste bloquée jusqu'à ce que la variable condition soit signalée et que l'activité ait réussi à réacquérir le verrou.

```
int pthread_cond_signal (pthread_cond_t *cond);
```

Signale la variable condition : une activité bloquée sur la variable condition est réveillée. Cette activité tente alors de réacquérir le verrou correspondant à son appel de `cond_wait`. Elle sera effectivement débloquée quand elle réussira à réacquérir ce verrou. Il n'y a aucun ordre garanti pour le choix de l'activité réveillée. L'opération signal n'a aucun effet s'il n'y a aucune activité bloquée sur la variable condition (pas de mémorisation).

```
int pthread_cond_broadcast (pthread_cond_t *cond);
```

Toutes les activités en attente sont réveillées, et tentent d'obtenir le verrou correspondant à leur appel de `cond_wait`.

Remarques : Les verrous sont par défaut des verrous d'exclusion mutuelle. En utilisant les attributs (non documentés ici), on peut créer un verrou dit récursif qui peut être verrouillé plusieurs fois par la même activité (le verrou doit alors être autant de fois déverrouillé avant qu'une autre activité puisse l'acquérir).
Par ailleurs, contrairement à la définition des moniteurs de Hoare, l'activité signalée n'a pas priorité sur le signaleur : le signaleur ne perd pas l'accès au moniteur s'il le possédait, et le signalé reste bloqué tant qu'il n'obtient pas le verrou. C'est pourquoi il est nécessaire d'utiliser une boucle d'attente réévaluant la condition d'exécution. En effet, cette condition peut être invalidée entre le moment où l'activité est signalée et le moment où elle obtient effectivement le verrou, par exemple si une autre activité obtient le mutex et pénètre dans le moniteur avant l'activité signalée.
Enfin, pour des raisons d'efficacité, il est courant de faire l'appel à `cond_signal` hors de la zone lock-unlock, de sorte que l'activité signalée puisse acquérir plus facilement le verrou. Attention cependant à garantir l'atomicité des opérations du moniteur !

Exemple : Considérons deux variables partagées `x` et `y`, protégée pat le mutex `mu` et la variable de condition `cond` qui doit être signalée lorsque `x`devient supérieure à `y`.

```
int x,y;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Attente jusqu'à ce qu'`x` soit supérieur à `y` est réalisée par :

```
pthread_mutex_lock(&mut);
while (x <= y)
{
pthread_cond_wait(&cond, &mut);
}
/* operate on x and y */
pthread_mutex_unlock(&mut);
```

Les modifications sur `x` et `y` qui pourraient faire que`x` deviennent supérieure à `y` doivent signaler la condition si nécessaire :

```
pthread_mutex_lock(&mut);
/* modify x and y */
if (x > y) pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mut);
```

S'il peut être prouvé qu'au moins un des threads appelant doit être réveillé (par exemple, s'il y a seulement 2 threads qui communique via `x` et `y`; `pthread_cond_signal` peut être utilisée comme une alternative plus efficace que `pthread_cond_broadcast`. Dans le doute on utilise `pthread_cond_broadcast`.
Pour attendre que `x` devienne supérieur à `y` avec un délai de 5 secondes, on fait :

```
struct timeval now;
struct timespec timeout;
int retcode;

pthread_mutex_lock(&mut);
gettimeofday(&now);
timeout.tv_sec = now.tv_sec + 5;
timeout.tv_nsec = now.tv_usec * 1000;
retcode = 0;
while (x <= y && retcode != ETIMEDOUT)
{
retcode = pthread_cond_timedwait(&cond, &mut, &timeout);
}
if (retcode == ETIMEDOUT)
{
/* timeout occurred */
    }
else
{
    /* operate on x and y */
    }
pthread_mutex_unlock(&mut);
```

Exercice : Voici un exemple simple :**condition.c**

```
//Exemple de moniteur : producteur/consommateur à une case

/*Compilation
gcc -D_REENTRANT -o exemple2 exemple2.c -lpthread
*/
#include <pthread.h>
#include <string.h>

static pthread_cond_t est_libre, est_plein;
static pthread_mutex_t protect;
static char *buffer;

/* Depose le message msg (qui est dupliqué). Bloque tant que le tampon est plein. */
void deposer (char *msg)
{
    pthread_mutex_lock (&protect);
    while (buffer != NULL)
        pthread_cond_wait (&est_libre, &protect);
    /* buffer = NULL */
    buffer = strdup (msg); /* duplication de msg */
    pthread_cond_signal (&est_plein);
    pthread_mutex_unlock (&protect);
}

/* Renvoie le message en tête du tampon. Bloque tant que le tampon est vide.
 * La libération de la mémoire contenant le message est à la charge de l'appelant. */
char *retirer (void)
```

```
{
    char *result;
    pthread_mutex_lock (&protect);
    while (buffer == NULL)
        pthread_cond_wait (&est_plein, &protect);
    /* buffer != NULL */
    result = buffer;
    buffer = NULL;
    pthread_cond_signal (&est_libre);
    pthread_mutex_unlock (&protect);
    return result;
}

/* Initialise le producteur/consommateur. */
void init_prodcons (void)
{
    pthread_mutex_init (&protect, NULL);
    pthread_cond_init (&est_libre, NULL);
    pthread_cond_init (&est_plein, NULL);
    buffer = NULL;
}

/*-------------------------------------------------------------------
  Fonction executee par les threads.
  -------------------------------------------------------------------*/
void Thread_ret (void)
{
  int i;
char message[25];
for (i=0;i<5;i++)
{
sprintf(message,"%d",i);
strcat(message, "--message--");
deposer(message);
printf("Depose message %d \n",i);
}                                                                    pthread_exit(NULL);
}

void Thread_dep (void)
{
  int i;
char * message;
for (i=0;i<5;i++)
{
message = retirer();
printf("Retire message %d \n",i);
printf("Les message est : %s\n", message);
sleep(1); /*pour voir la synchronisation*/
}                                                                    pthread_exit(NULL);
}
/*-------------------------------------------------------------------*/


int main(int argc, char *argv[])
{
 pthread_t thr_main, threads[2];
int i, NbThreads;
 thr_main = pthread_self();
NbThreads = 2;

/*Initialisation du producteur/consommateur*/
```

```
init_prodcons();

 /* creation des threads  */
 pthread_create(&threads[0], NULL, (void *)Thread_dep, NULL);
printf("------ Thr %d (main) cree: %d (i = %d)\n",(int)thr_main,(int)threads[0],0);

 pthread_create(&threads[1], NULL, (void *)Thread_ret, NULL);
printf("------ Thr %d (main) cree: %d (i = %d)\n",(int)thr_main,(int)threads[1],1);

/*thread en mode join*/
pthread_join(threads[0], NULL);
       printf("------main thread %d (main) fin de : %d\n",
             (int)thr_main, (int)threads[0]);

pthread_join(threads[1], NULL);
       printf("------main thread %d (main) fin de : %d\n",
             (int)thr_main, (int)threads[1]);
printf("------main Fin de main (%d)\n", (int)thr_main);
return 0;
}
```

Compiler le programme.
Que se passe-t-il ? Pourquoi ?
Réponse :

12.8.3 Les sémaphores POSIX

Les sémaphores sont des compteurs pour des ressources partagées entre les threads. Les opérations de bases sur les sémaphores sont : l'incrémentation atomiquement, et attente jusqu'à ce que le compteur soit non nul et décrémentation atomiquement.

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

`sem_init` initialise l'objet sémaphore pointé par `sem`. Le compteur associé au sémaphore est fixé initialement à`value`. L'argument `pshared` indique si le sémaphore est local au processus courant (`pshared` à 0) ou s'il est partagé par plusieurs processus (`pshared` différent de 0). LinuxThreads ne supporte pas pour l'instant les sémaphores partagés, donc `sem_init` retourne toujours avec l'erreur `ENOSYS` si `pshared` est différent de 0.

```
int sem_wait(sem_t * sem);
```

`sem_wait` suspend le thread appelant jusqu'à ce que le sémaphore pointé par `sem` ait un compteur non nul. Alors le compteur du sémaphore est atomiquement décrémenté.

```
int sem_trywait(sem_t * sem);
```

`sem_trywait` est la variante non bloquante de `sem_wait`. Si le sémaphore pointé par `sem` possède un compteur non nul, le compteur est décrémenté atomiquement et `sem_trywait` retourne immédiatement 0. Si le sémaphore possède un compteur nul, `sem_trywait` retourne immédiatement l'erreur `EAGAIN`.

```
int sem_post(sem_t * sem);
```

`sem_post` incrémente atomiquement le compteur du sémaphore pointé par `sem`. Cette fonction n'est jamais bloquante et peut être utilisée de manière sure par des gestionnaires de signaux asynchrones.

```
int sem_getvalue(sem_t * sem, int * sval);
```

`sem_getvalue` stocke à l'endroit pointé par `sval` le compteur courant du sémaphore `sem`.

```
int sem_destroy(sem_t * sem);
```

`sem_destroy` détruit un objet sémaphore, en libérant les ressources qu'il possédait. Aucun threads ne doit attendre le sémaphore lorsque celui-ci est détruit. Dans l'implémentation LinuxThreads, aucune ressources n'est associée à l'objet sémaphore, donc, pour l'instant, la fonction `sem_destroy` ne fait rien à part tester si aucun thread n'attend après l'objet sémaphore.
L'utilisation de sémaphores permet aussi la synchronisation entre plusieurs *threads*. Voici un exemple simple :**thread3.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
static sem_t my_sem;
int the_end;
void *thread1_process (void * arg)
{
  while (!the_end) {
    printf ("Je t'attend !\n");
    sem_wait (&my_sem);
  }
  printf ("OK, je sors !\n");
  pthread_exit (0);
}
void *thread2_process (void * arg)
{
  register int i;
  for (i = 0 ; i < 5 ; i++) {
    printf ("J'arrive %d !\n", i);
    sem_post (&my_sem);
    sleep (1);
  }
  the_end = 1;
  sem_post (&my_sem); /* Pour debloquer le dernier sem_wait */
  pthread_exit (0);
}
main (int ac, char **av)
{
  pthread_t th1, th2;
  void *ret;
  sem_init (&my_sem, 0, 0);
  if (pthread_create (&th1, NULL, thread1_process, NULL) < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
```

```
  }
  if (pthread_create (&th2, NULL, thread2_process, NULL) < 0) {
    fprintf (stderr, "pthread_create error for thread 2\n");
    exit (1);
  }
  (void)pthread_join (th1, &ret);
  (void)pthread_join (th2, &ret);
}
```

Compiler le programme.
Que se passe-t-il ? Pourquoi ?
Réponse :

12.9 Mode de création des *threads* : `JOINABLE` ou `DETACHED`

Dans les exemples précédents, les *threads* sont créés en mode `JOINABLE`, c'est à dire que le processus qui a créé le *thread* attend la fin de celui-ci en restant bloqué sur l'appel à `pthread_join`. Lorsque le *thread* se termine, les resources mémoire du *thread* sont libérées grâce à l'appel à `pthread_join`. Si cet appel n'est pas effectué, la mémoire n'est pas libérée et il s'en suit une fuite de mémoire. Pour éviter un appel systématique à `pthread_join` (qui peut parfois être contraignant dans certaines applications), on peut créer le *thread* en mode `DETACHED`. Dans ce cas la, la mémoire sera correctement libérée à la fin du *thread*.
Pour cela il suffit d'ajouter le code suivant :

```
  pthread_attr_t thread_attr;
  if (pthread_attr_init (&thread_attr) != 0) {
    fprintf (stderr, "pthread_attr_init error");
    exit (1);
  }
  if (pthread_attr_setdetachstate (&thread_attr, PTHREAD_CREATE_DETACHED) != 0) {
    fprintf (stderr, "pthread_attr_setdetachstate error");
    exit (1);
  }
```

puis de créer les *threads* avec des appels du type :

```
  if (pthread_create (&th1, &thread_attr, thread1_process, NULL) < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }
```

Écrire le programme en reprenant **thread3.c**, et le compiler.
Que se passe-t-il ? Pourquoi ?
Réponse :

12.10 Destruction de *thread* : cancellation

Les exemples ci-dessus utilisaient la fonction `pthread_exit` pour la destruction d'un *thread* (en fait le *thread* se détruisait tout seul). Il existe un mécanisme dans lequel un *thread* peut en détruire en autre à condition que ce dernier ait validé cette possibilité. Le comportement par défaut est de type décalé (`deferred`). Lorsqu'on envoie une requête de destruction d'un *thread*, celle-ci n'est exécutée que lorsque ce *thread* passe par un cancellation point comme par exemple l'appel à la fonction `pthread_testcancel`. Voici un petit exemple : **thread4.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *my_thread_process (void * arg)
{
  int i;
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
    pthread_testcancel (); /*ICI, permis de tuer = license to kill*/
  }
}
main (int ac, char **av)
{
  pthread_t th1, th2;
  void *ret;
  if (pthread_create (&th1, NULL, my_thread_process, "1") < 0) {
    fprintf (stderr, "pthread_create error for thread 1\n");
    exit (1);
  }
  sleep (2);
  if (pthread_cancel (th1) != 0) {
    fprintf (stderr, "pthread_cancel error for thread 1\n");
    exit (1);
  }
  (void)pthread_join (th1, &ret);
}
```

Compiler le programme.
Que se passe-t-il ? Pourquoi ?

Réponse :

Pour éviter l'utilisation des cancellation points, on peut indiquer que la destruction est en mode asynchrone en modifiant le code du *thread* de la manière suivante :

```
void *my_thread_process (void * arg)
{
  int i;
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
  for (i = 0 ; i < 5 ; i++) {
    printf ("Thread %s: %d\n", (char*)arg, i);
    sleep (1);
  }
}
```

Écrire le programme en reprenant **thread4.c**, et le compiler.
Que se passe-t-il ? Pourquoi ?

Réponse :

12.11 Excercice 1

Dans cet exercice, trois *threads* seront créés, et ils incrémenteront chacun une variable globale différente. La valeur de chacune de ces variables est affichée par le programme initial.

Réponse :

12.12 Excercice 2

Dans cet exercice, il s'agit de mettre en pratique la communication entre threads via une variable de condition.

Dans notre exercice, 5 threads vont accéder à une ressource de 7 emplacements. Chacun des threads créés va demander successivement 3 puis 2 puis 1 emplacements de la ressource. Après chaque accès aux ressources le thread attend une seconde et libère le ou les emplacements accédés.

Il faut bien entendu mettre en attente active tout thread qui désire accéder à un ou plusieurs emplacement alors que toutes les ressources sont accédées. Il faut donc mettre en place un mécanisme d'exlusion.

Réponse :

12.13 Debug d'un programme multi-thread sous LINUX

Les dernières versions de `gdb` et de la `glibc` permettent de debugger un programme LINUX utilisant du multi-threading. Plus d'infos sont disponibles sur la page WWW de la bibliothèque LinuxThreads (voir bibliographie). Voici

un petit exemple de session gdb sur le programme d'exemple `thread1`. La compilation s'effectue de la manière suivante :
`gcc -ggdb -D_REENTRANT -o thread1 thread1.c`

```
(gdb) b main
Breakpoint 1 at 0x8048622: file thread1.c, line 22.
```

On a posé un point d'arrêt dans le programme principal avant la création des *threads*.

```
(gdb) n
[New Thread 25572]
[New Thread 25571]
[New Thread 25573]
Thread 1: 0
Thread 1: 1
Thread 1: 2
Thread 1: 3
Thread 1: 4
(gdb) info threads
  3 Thread 25573  0x4007a921 in __libc_nanosleep ()
* 2 Thread 25571  main (ac=1, av=0xbffffca0) at thread1.c:27
  1 Thread 25572  0x4008b2de in __select ()
```

L'action sur next exécute la fonction `pthread_create` qui provoque la création du thread 1. La commande info *threads* permet de connaître la liste des tous les *threads* associés à l'exécution du programme. Le *thread* courant est indiqué par l'étoile, le numéro du *thread* est indiqué en deuxième colonne (ici 1, 2, 3).

```
(gdb) thread 1
[Switching to Thread 25654]
#0  0x4008b2de in __select ()
```

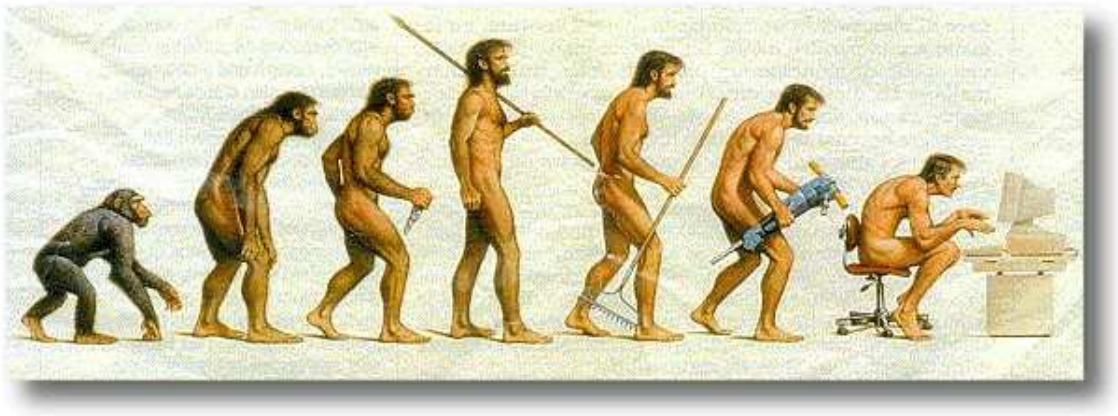
On peut passer d'un *thread* à l'autre en utilisant la commande *thread* numéro-du-thread.

12.14 Conclusion et bibliographie

L'utilisation du multi-threading permet de faciliter la programmation d'un grand nombres d'applications de type serveur ou multimédia, tout en améliorant les fonctionnalités du programme par rapport à une solution classique basée sur l'utilisation des créations de processus (`fork`). Les pointeurs suivants vous seront utiles si vous vous lancez dans le multi-threading :

- La bibliothèque LinuxThreads sur http://pauillac.inria.fr/~xleroy/linuxthreads
- Le site "Programming POSIX *threads*" sur http://www.humanfactor.com/pthreads
- Si vous voulez porter vos applicatifs sur Win32, la bibliothèque "POSIX Threads (pthreads) for Win32" sur http://sourceware.cygnus.com/pthreads-win32
- La FAQ du groupe de discussion `comp.programming.threads` sur http://www.serpentine.com/~bos/threads-faq

Vous en avez maintenant terminé avec Unix...



Réflexion sur l'évolution humaine...

Cette page est laissée blanche intentionnellement

Bibliographie

[Agi95] Axis & Agix. *Unix Utilisation : Guide de Formation.* EDITIONS LASER, 1995.

[Bla02a] CH. Blaess. *Langages de scripts sous Linux.* Eyrolles, 2002.

[Bla02b] CH. Blaess. *Programmation Système en C sous Linux.* Eyrolles, 2002.

[CH95] J.M. Champarnaud and G. Hansel. *Passeport pour UNIX et C.* Passeport pour l'informatique. Internationnal Thomson Publishing, 1995.

[Cha94] P. Charman. *Unix et X-Window : Guide Pratique.* CÉPADUÈS ÉDITIONS, 1994.

[Pou94] T. Poulain. Cours unix. 1994.

[Sto92] Richard Stoeckel. *Filtres et Utilitaires Unix.* ARMAND COLIN, 1992.