Le système Unix

Sylvie Borne et Lucas Létocart
LIPN - UMR CNRS 7030
Institut Galilée, Université Paris 13
99 av. Jean-Baptiste Clément
93430 Villetaneuse - FRANCE
{sylvie.borne,lucas.letocart}@lipn.univ-paris13.fr

Chapitre Contenu du document.

Chapitre 1. Quelques généralités sur le système Unix

1.1. Historique

Le système Unix a été développé à la fin des années 60 par une équipe des laboratoires Bell de AT&T, dirigée par Ken Thompson. Initialement écrit en langage assembleur, le système Unix a été réécrit au début des années 70 dans un langage de haut niveau spécialement conçu pour : le langage C. En 1979 apparaît la version 7 ou System III, véritable origine de l'Unix commercial. Depuis, deux grandes familles de systèmes Unix coexistent :

- ✓ Le système V d'AT&T,
- ✓ Le système BSD de l'Université de Berkley.

Il existe de nombreuses versions dérivées. On distingue :

Les "Unix based" : dérivés des sources AT&T et/ou Berkeley par la firme qui en a fait l'acquisition comme Solaris de Sun, AIX d'IBM, Linux, ...

Les "Unix like" (tendent à disparaître !) : systèmes dont le comportement est similaire à Unix mais réécrits (IDRIS, UNOX, ROS, ...) et dédiés à des applictions particulières (temps réel, transactionnel, ...)

Deux organismes de normalisation (à l'origine groupements d'utilisateurs) s'attachent à définir une norme internationale pour Unix : le groupe X/Open (1984) et le groupe **POSIX** (1990).

Sous le système Unix, il existe un grand nombre de logiciels du domaine public (logiciels de la **Free Software Foundation** - organisation pour la gratuité des logiciels sous UNIX) : outils GNU (compilateurs C, C++, make, awk, emacs), Linux (conforme au standard POSIX).

1.2. Spécificités d'Unix

Le système Unix est un système d'exploitation multi-utilisateurs et multi-tâches

Concernant l'aspect multi-utilisateurs, précisons qu'il existe deux types d'utilisateurs :

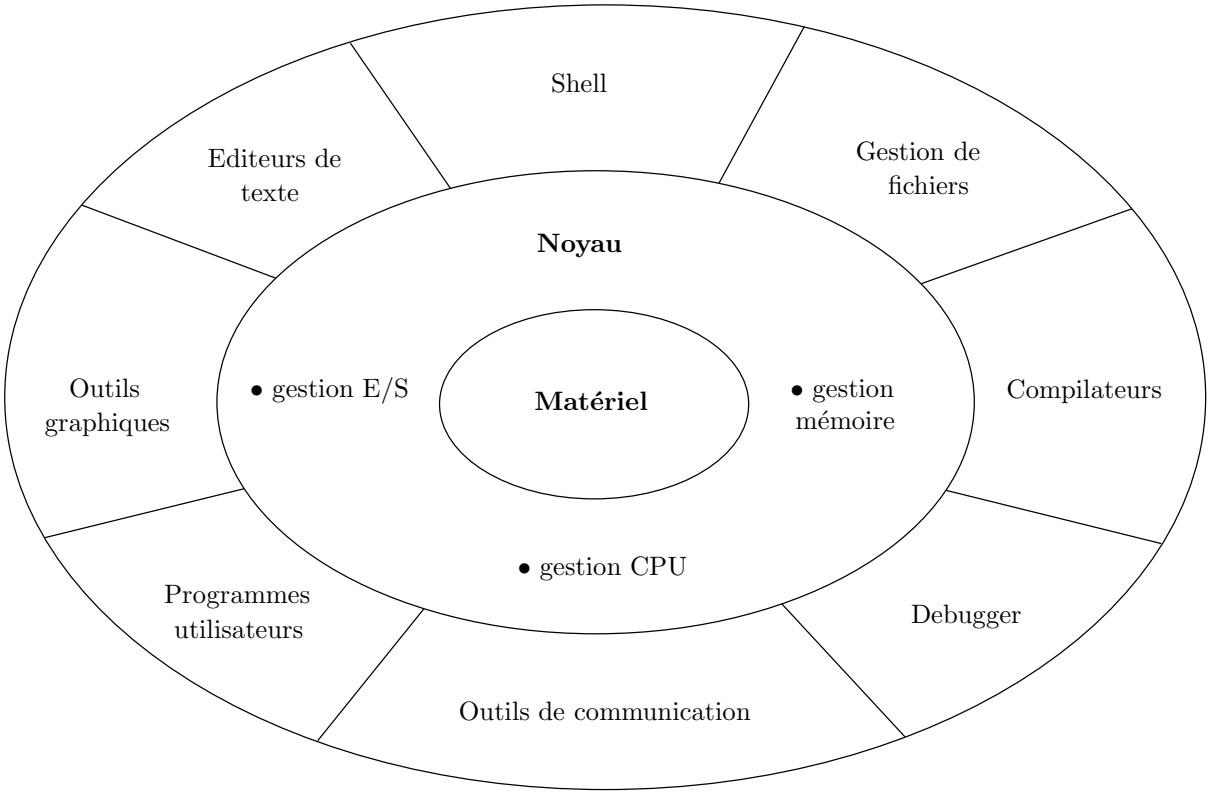
- ✓ Les utilisateurs "normaux" : ont un compte avec un nom de connexion (login name), un mot de passe (password), un espace de travail protégé sur le disque (un répertoire privé - home directory - dans le système de fichiers), une boîte aux lettres (mail box).
- ✓ Le super-utilisateur est chargé de gérer le système, c'est-à-dire de veiller au bon usage par l'ensemble des utilisateurs des ressources partagées de la machine (il dispose de privilèges particuliers et d'un compte à part appelé root).

Unix est multi-tâches car plusieurs programmes peuvent être en cours d'exécution en même temps sur une même machine. En fait, à chaque instant, le processeur ne traite qu'au plus un seul des programmes lancés.

Le système Unix est également le système privilégié pour la communication entre ordinateurs et entre utilisateurs.

1.3. Architecture en couches

Le schéma ci-dessous est une représentation des composants logiciels constituant le système Unix :



Le **noyau** est la couche logicielle la plus interne du système Unix. Elle est dédiée à la gestion des composants matériels : processeur, mémoire, périphériques. Autour du noyau gravite un certain nombre d'utilitaires. Le **shell** est l'interpréteur de commande Unix qui vérifie et interprète les commandes, les exécute et renvoie les réponses.

1.4. Ouvrir et fermer une session Unix

Pour ouvrir une session, il faut entrer son nom de login et son mot de passe :

login : `letocart`
passwd :
`$`

Pour fermer une session :

`$` **exit** ou **ctrl/d**

1.5. Les commandes Unix

Une commande Unix est un nom d'un fichier contenant un programme exécutable (il existe également des commandes internes au shell).

Pour lancer une commande, la syntaxe est la suivante :
`nom_commande [-liste_options] [liste_arguments]`

Exemple :
`wc -lw fich1 fich2 fich3`

L'option permet de modifier le comportement de la commande. Le ou les argument(s) peuvent être une expression, un fichier ou un processus.

1.6. Le manuel en ligne d'Unix

man `com` : fournit des informations sur la commande unix `com`.

Chapitre 2. Le système de fichiers

2.1. Le fichier Unix

2.1.1. Définition

Sous Unix, un fichier est un objet recevant et délivrant des données, constitué d'une chaîne non structurée de caractères. On distingue trois types de fichiers :

- ✓ Le fichier ordinaire qui est un ensemble de données stocké sur un disque ou un dispositif analogue (cd-rom, bande,...),
- ✓ Le répertoire qui est un ensemble d'informations permettant d'accéder à d'autres fichiers,
- ✓ Le fichier spécial qui représente un dispositif physique d'entrées/sorties (terminal, imprimante, lecteur de bande et de disquettes,...).

Un fichier est décrit dans une structure de données appelée i-nœud (ou inode) comportant les informations suivantes :

- type de fichier (répertoire, fichier ordinaire, ...)
- mode de protection,
- nombre de liens,
- numéro du propriétaire,
- numéro du groupe,
- taille du fichier en octets,
- adresses physiques directes,
- adresses physiques indirectes,
- date et heure du dernier accès,

- date et heure de la dernière modification du fichier,
- date et heure de la dernière modification de l'i-nœud.

Les i-nœuds sont rangés dans une table où chaque i-nœud a un numéro (index).

Le nom du fichier n'est pas une information stockée dans le i-nœud mais dans le répertoire auquel appartient ce fichier. Ainsi, un répertoire est un fichier contenant une suite de couples (index de i-nœud, nom). Un tel couple est appelé un **LIEN**.

2.1.1. Opérations élémentaires sur les fichiers

Affichage du contenu d'un fichier

cat *fich...*	Affiche sur la sortie standard le contenu des fichiers donnés en argument
more *fich...*	Affiche page par page sur la sortie standard le contenu des fichiers donnés en argument

Affichage du contenu d'un répertoire

ls *fich...*	Affiche sur la sortie standard le contenu des répertoires donnés en argument

2.2. Arborescence de fichiers

L'ensemble des fichiers gérés par le système Unix est organisé en une seule arborescence. Au sein de cette arborescence, on retrouve classiquement les répertoires suivants :

/ $\Leftrightarrow$ racine de l'arborescence

bin $\subset$ fichiers binaires exécutables = commandes du système

lib $\subset$ librairies pour la programmation

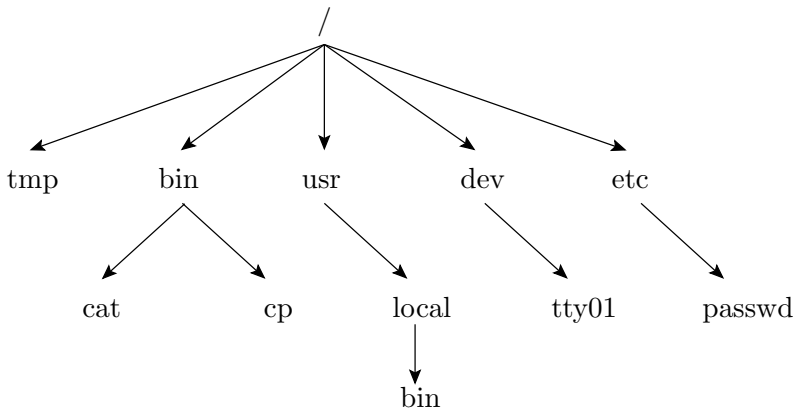
etc $\subset$ fichiers de contrôle pour l'administration du système

dev $\subset$ fichiers spéciaux représentant les périphériques

usr $\subset$ fichiers dédiés aux utilisateurs

tmp $\subset$ fichiers temporaires utilisés par les éditeurs, les compilateurs et autres utilitaires.

Voici un exemple d'arborescence de fichiers :



Chaque utilisateur possède un répertoire privé.

Pour accéder à un fichier :

- Référence absolue : chemin à partir de la racine

  `/usr/local/bin`

- Référence relative : chemin à partir du répertoire de travail

  . répertoire de travail

  .. répertoire "père"

Pour se déplacer dans l'arborescence

pwd (print working directory) : indique la référence absolue du répertoire de travail

cd (change directory)

ls -R : liste récursivement les sous-répertoires et leur contenu

À sa création (avec la commande `mkdir`), tout répertoire contient deux liens :

(index,.) : un lien sur lui-même,

(index,..) : un lien sur son père.

À chaque fichier physique ordinaire ou spécial, on peut associer un ou plusieurs liens dans différents répertoires de l'arborescence. Ces différents couples (index, nom) ont tous le même index de i-nœud, mais ont des noms nécessairement différents si ces liens appartiennent au même répertoire. Un fichier peut être supprimé (et l'espace disque qu'il occupait récupéré par le système) si et seulement si le nombre de liens sur ce fichier est égal à 0.

Le nombre de liens sur un répertoire est supérieur ou égal à 2 :

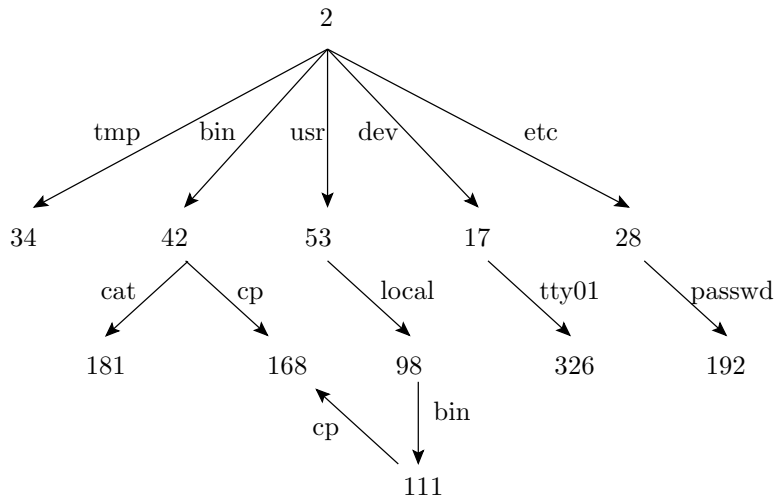
→ un premier lien (index, nom) dans le répertoire père,

→ un deuxième lien (index,.) dans lui-même,

→ et éventuellement d'autres liens (index,..) dans ses sous-répertoires.

Seul le répertoire / est un fichier sans nom, d'index 2 disposant d'une adresse fixe sur le disque connue par le système au démarrage.

Il est donc possible de représenter différemment l'arborescence de fichiers sous Unix en représentant les liens comme ci-dessous :



2.3. Quelques commandes pour réaliser des opérations élémentaires sur les fichiers

cp `fich_source fich_dest`	Recopie physique de `fich_source` en `fich_dest`
ln `fich_source fich_dest`	Attribue un lien supplémentaire sur `fich_dest`, `fich_dest` et `fich_source` ont le même i-nœud
rm `fich`	Suppression du lien fich
rmdir `rep`	Suppression du répertoire `rep` (il doit être vide)
mv `fich_source fich_dest`	Renommer le fichier `fich_source` en `fich_dest`

2.4. "File system"

On a jusqu'à présent considéré que le système ne gérait qu'un seul disque physique et que les index des i-nœuds étaient tous différents. En fait, il peut exister différents disques, chacun d'eux pouvant être partitionné entre différents disques logiques.

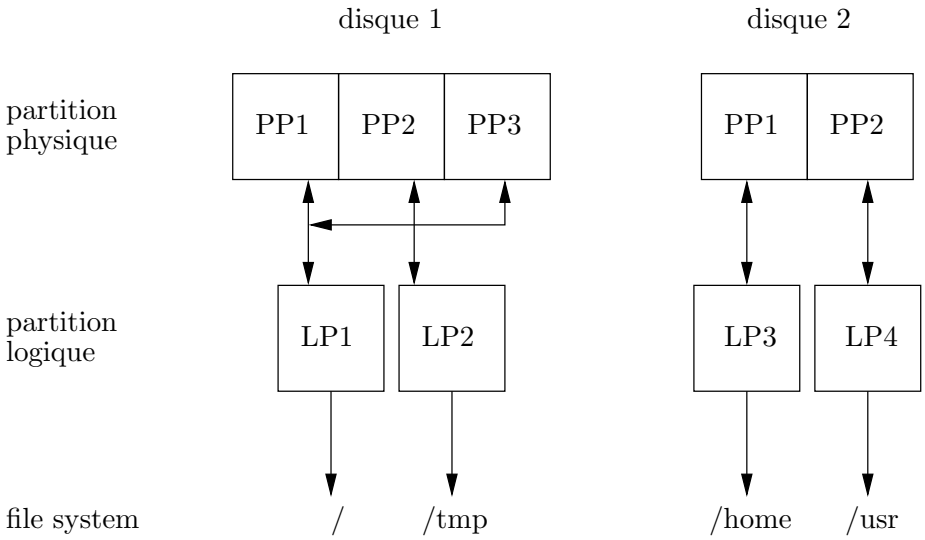
2.4.4. Définition

L'espace disque physique disponible peut être constitué d'un ou plusieurs disques, chacun pouvant être divisé en une ou plusieurs partitions logiques.

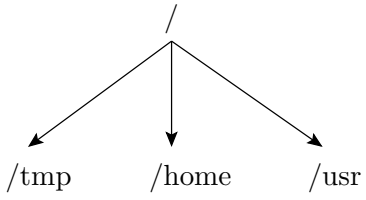
Un file system est un système de fichiers sur un disque logique (il possède donc sa propre table de i-nœuds).

L'arborescence globale de fichiers est en fait constituée de plusieurs "file system" montés (commande **mount**). On dit qu'un file system est monté lorsqu'un répertoire de l'arborescence est en réalité un répertoire racine d'un disque logique.

Le root file system est le disque système contenant la racine de l'arborescence, le noyau, les fichiers système, ...



Arborescence :



2.4.4. Structure de base d'un file system

Un file system est un ensemble de blocs dont la taille est choisie à l'initialisation du file system (de 512 octets à plus de 8192).

bloc	contenu
0	boot bloc
1	super bloc
2 ... k	table des inodes
k+1 ...	bloc de données

Bloc 0 = boot + identification du disque

Bloc 1 = informations sur le file system (date de dernière mise à jour, taille, pointeur sur la liste des blocs libres, nombre max de fichiers)

Bloc 2 à k = table des inodes

A partir du bloc k+1 = données

La commande **df** indique le nombre de blocs disponibles par file system.

2.4.4. Liens symboliques

Il s'avère donc que des fichiers physiques différents appartenant à des disques logiques distincts peuvent avoir le même index de i-nœud. Il n'est donc pas possible de créer des liens entre différents file system. Le système Unix permet alors de créer des **liens symboliques** entre des fichiers appartenant à différents file system.

Un lien symbolique est un fichier contenant la référence absolue d'un autre fichier. Toute opération sur ce fichier (lecture, écriture...) s'effectue sur le fichier référencé.

Commande :

ln -s `fich_source fich_dest`

Crée un lien symbolique `fich_dest` contenant la référence à `fich_source`.

2.5. Protection des fichiers

On appelle mode de protection un ensemble d'autorisations sur un fichier déterminant **qui** peut avoir accès à un fichier et en vue de **quelle** utilisation.

Qui

- ➢ le propriétaire : **u** pour user
- ➢ les membres du groupe du propriétaire : **g** pour group
- ➢ les autres utilisateurs : **o** pour others

Quelle

	fichier ordinaire ou spécial	répertoire
lecture (=4) ⇔ **r** pour read	lecture du fichier	liste le contenu d'un répertoire
écriture(=2) ⇔ **w** pour write	modification du fichier	création ou suppression de liens dans le répertoire
exécution(=1) ⇔ **x** pour execute	exécution d'un fichier exécutable	accès aux fichiers et aux sous-répertoires contenus dans le répertoire

Le mode de protection d'un fichier peut être représenté par une chaîne de 9 caractères (3 permissions possibles pour chaque type d'utilisateurs), par exemple,

```
rwxrw-r-- fich
```

ou, par trois chiffres (somme des valeurs de permissions accordées à chaque type d'utilisateurs)

```
764 fich
```

Seul le propriétaire d'un fichier peut modifier le mode de protection d'un fichier. Le super-utilisateur (root) a tous les droits sur tous les fichiers.

Commandes pour définir un mode de protection :

chmod : changer le mode de protection d'un fichier

Syntaxe 1 (mode absolu) :

```
chmod 764 fich
```

Syntaxe 2 (mode symbolique) :

```
chmod a=r,u+w fich
```

Ecriture d'expression contenant au moins 3 éléments :

- le destinataire de la commande (`u`, `g`, `o` et `a` pour tous les utilisateurs)
- un opérateur : `+` pour l'ajout d'une ou de plusieurs permissions, `-` pour le retrait, `=` pour l'assignation de permissions
- la permission : `r`, `w` et `x`.

umask : attribuer des permissions par défaut aux fichiers créés après l'usage de la commande

```
umask 033 ↩
```

permissions par défaut : `rwxr--r--`

```
umask ↩
```

Consultation du mode en cours.

Chapitre 3. Les processus

La différence essentielle entre Unix et Dos est la suivante : sous Unix, plusieurs programmes peuvent s'exécuter simultanément, le système se chargeant de leur affecter tour à tour le processeur.

3.1. Définition

Soit P1, P2, ..., Pn, n programmes exécutés en même temps. Le système va, pendant un bref laps de temps, exécuter l'un d'eux, par exemple Pi, sur le processeur. Si Pi se termine, l'exécution de Pi+1 est lancée, sinon Pi est mis en attente par le système pour permettre d'exécuter Pi+1 pendant un nouveau laps de temps. En conséquence, un programme ne s'exécute pas en continu durant une certaine période. Le cycle d'exécution d'un programme se décompose en un grand nombre de périodes d'activité et d'attente. Pour gérer cette activité multi-tâche, le système a besoin d'avoir en mémoire des informations sur les programmes en exécution. Un **PROCESSUS** est un programme en cours d'exécution.

On distingue deux types de processus :

- Le processus système (daemons) : assure des services généraux accessibles à tous les utilisateurs du système. Le propriétaire est le super-utilisateur et il n'est sous le contrôle d'aucun terminal.
- Le processus utilisateur : dédié à l'exécution d'un tâche particulière. Le propriétaire est l'utilisateur qui l'exécute et il est sous le contrôle du terminal à partir duquel il a été lancé.

3.2. Cycle de vie d'un processus

3.2.2. Création

Toute exécution d'un programme déclenche la création d'un processus dont la durée de vie est égale à la durée d'exécution du programme.

Le système alloue à chaque processus un numéro d'identification unique : **PID** (Process IDentifier).

Tout processus est créé par un autre processus : son processus père.

Exemple : Lorsqu'un utilisateur lance une commande, un processus est créé dont le père est le processus correspondant à l'exécution du shell.

3.2.2. États d'un processus

- Prêt (R) : le processus attend que le processeur lui soit affecté.
- Actif (R) : le processeur exécute le processus
  → mode utilisateur : exécution des instructions du programme
  → mode système (ou noyau) : exécution d'un appel système (non interruptible).
- Endormi (S pour <20s ou I pour >20s) : le processus est en attente de l'arrivée d'un évènement (par exemple, une réponse du terminal).
- Zombi (W) : le processus est terminé.
- Suspendu (T) : le processus a été interrompu et attend l'arrivée d'un signal de reprise.

3.2.2. Politique de swapping

Chaque processus présent dans le système occupe différentes régions en mémoire centrale. Quand un processus a besoin pour son exécution d'une taille de mémoire supérieure à celle disponible, le système tente de libérer de la place mémoire en copiant en zone de débordement (swap) les zones occupées par certains processus en attente. La zone de swap est une partition du disque.

3.2.2. La commande ps

La commande **ps** donne la liste des processus en cours. Sans option, il s'agit de la liste des processus de la connexion en cours appartenant à l'utilisateur avec les informations suivantes :

- **PID** : numéro de processus
- **TTY** : terminal qui contrôle le processus (? s'il s'agit d'un processus sans terminal de rattachement)
- **TIME** : temps cumulé d'exécution du processus et de sa descendance
- **CMD** : commande exécutée

3.3. Les fichiers standards et les redirections

Tout processus communique avec l'extérieur par le biais de trois fichiers logiques appelés fichiers standard d'entrées/sorties :

- ➢ Le fichier entrée standard (stdin de numéro 0) sur lequel le processus lit ses données,
- ➢ Le fichier sortie standard (stdout de numéro 1) sur lequel le processus écrit ses données,
- ➢ Le fichier sortie erreur standard (stderr de numéro 2) sur lequel le processus écrit ses messages d'erreur.

Quand un processus naît, ces trois fichiers sont ouverts. Par défaut, ils sont associés au terminal de lancement : le clavier pour le fichier d'entrée standard, l'écran pour la sortie standard et pour la sortie erreur standard.

Il est possible de rediriger les E/S standards d'un processus c'est-à-dire de leur associer un fichier autre que le terminal. Pour cela, on utilise la syntaxe suivante :

`commande < fich` : redirige l'entrée standard de `commande` sur le fichier `fich` (qui doit exister et être lisible par l'utilisateur). Ceci signifie que `commande` lit ses données en entrée dans le fichier `fich`.

`commande > fiché` : redirige la sortie standard de `commande` sur le fichier `fich` (avec écrasement de celui-ci s'il existe et création s'il n'existe pas).

`commande >> fich` : redirige la sortie standard de `commande` sur le fichier `fich` sans écrasement mais en ajoutant).

`commande >& fich` : redirige (en C-shell) la sortie standard et la sortie erreur standard de `commande` sur le fichier `fich`.

Exemple :
Le résultat de la commande `ls` est redirigé vers le fichier `commandes` (le fichier est créé) :

```
%ls /bin > commandes
```

Le résultat de la commande `ls` est redirigé vers le fichier `commandes` le résultat est ajouté au contenu précédent) :

```
%ls /usr/local/bin >> commandes
```

L'entrée et la sortie standard de `wc` sont redirigées respectivement vers le fichier `commandes` et le fichier `nbre_commandes` :

```
%wc < commandes > nbre_commandes
%cat nbre_commandes
927 927 6821
```

En redirigeant la sortie standard de la commande `cat`, il est possible de concaténer des fichiers :

```
%cat commandes > nbre_commandes > fich
```

et de créer un fichier :

```
%cat > essai
Ceci est un fichier
créé sans l'aide d'un editeur
<ctrl-d>
```

3.4. Enchaînement de processus

Il est possible sur une même ligne de commande de lancer plusieurs processus.

➢ Lancement en séquence de plusieurs commandes

`%date ;who ;ps > resu`

Dans la commande précédente, il y a création d'un processus qui exécute la commande date. Le système attend la terminaison de ce processus pour créer un nouveau processus qui exécute la commande `who` ... Ces différents processus ne co-existent pas. Ainsi, seule la sortie standard de la commande `ps` est redirigée vers le fichier `resu`. Pour que les sorties standards de ces trois commandes soient redirigées vers le fichier `resu`, il faut écrire :

`%(date ;who ;ps) > resu`

➢ Lancement concurrent de processus communiquant par un tube (pipe). Il est possible de lancer l'exécution simultanée de plusieurs processus échangeant des données par l'intermédiaire de zones mémoires (gérées par le système) appelées tubes.

Syntaxe : `com1 | com2 | ... | comn`
Le système crée n-1 tubes et n processus :

- Le processus qui exécute `com1` a sa sortie standard redirigée vers le premier tube, son entrée standard étant celle définie par défaut, à savoir le clavier,
- Le processus qui exécute `com2` a son entrée standard redirigée vers le premier tube et sa sortie standard redirigée vers le deuxième tube,
- ...
- Le processus qui exécute `comn` a son entrée standard redirigée vers le n-1ième tube, sa sortie standard étant celle définie par défaut, à savoir l'écran.

Les n processus coexistent et se partagent l'accès au processus. Le système se charge de leur synchronsation : les processus lecteurs sont mis en attente tant que leur tube en entrée est vide, les processus écrivains sont mis en attente si leur tube en sortie est plein.

Exemple :
`%ls /bin /usr/local/bin | grep "cp" | wc -l`
Cette commande compte le nombre de fichiers dont le nom contient la chaîne `cp` dans les répertoires `/bin` et `/usr/local/bin`.

La commande `tee` permet de rediriger la sortie standard d'un processus vers un fichier et vers la sortie standard.
`%ls /bin /usr/local/bin | grep "cp" | tee fich_trie | wc -l`

Remarque : une cascade de la forme `com1 | com2 | ... | comn` ne présente vraiment d'intérêt que si les commandes `com1, com2`, ... et `comn` sont des filtres c'est-à-dire écrivent sur leur sortie standard et lisent sur leur entrée standard. Par exemple, `cat`, `wc` et `grep` sont des filtres alors que `echo`, `ls` ou `ps` n'en sont pas.

3.5. Lancement de processus en arrière-plan

Il est possible d'exécuter un programme tout en continuant à utiliser l'interpréteur de commandes à partir duquel ce programme a été lancé. Il suffit pour cela de lancer l'exécution du programme en arrière plan, ce qui est le cas lorsque la commande est suivie du caractère &.

Exemple :

```
%gcc infini.c -o infini
%infini &
[1] 360
```

Le processus associé à l'exécution du programme `infini` s'éxécute en arrière plan. L'interprète shell répond en indiquant le PID du processus créé (ici 360).

3.6. Les signaux

Au cours de l'exécution d'un processus, il est possible d'agir sur son déroulement en lui envoyant un signal. Unix définit de façon standard un certain nombre de signaux (cf. fichier `signal.h`) qui possèdent chacun une signature très précise dont :

- SIGINT (signal 2) : interrompre l'exécution d'un processus,
- SIGQUIT (signal 3) : interrompre l'exécution d'un processus et créer un fichier nommé core sur le disque contenant l'image mémoire du processus au moment de son arrêt,
- SIGKILL (signal 9) : arrêter définivement l'exécution d'un processus (ce signal ne peut pas être ignoré par le processus),
- SIGTSTP (signal 24) : suspendre temporairement l'exécution d'un processus,
- SIGCONT (signal 25) : reprendre l'exécution d'un processus précédemment suspendu par l'envoi d'un signal SIGTSTP.

Un signal peut être envoyé par :

- le système (c'est le cas par exemple des signaux d'erreur),
- un autre processus,
- l'utilisateur : soit l'utilisateur tape des caractères (pour les connaître, commande `stty -?` provoquant l'envoie d'un signal (par exemple, `ctrl z` pour SIGTSTP, `ctrl c` pour SIGINT, `ctrl ?` pour SIGQUIT) au processus en cours d'exécution sur le terminal ; soit l'utilisateur utilise la commande `kill` pour envoyer un signal à un ou plusieurs processus lorsqu'il n'a pas accès au terminal de rattachement des processus ou, lorsque ces derniers sont exécutés en arrière plan. La syntaxe est la suivante :
  `kill -signal PID...`
  `signal` désigne, soit le nom symbolique du signal, soit son numéro. La commande `ps` permet de connaître les PID des processus à qui l'on souhaite envoyer un signal.

Chapitre 4. Les interpréteurs de commandes

4.1. Définition

L'interpréteur de commandes, le shell, a pour rôle de traduire les commandes saisies par l'utilisateur afin que le système puisse les exécuter. On distingue deux types de commandes : celles qui sont internes au shell et dont l'exécution ne nécessite pas la création d'un processus, et celles qui sont externes au shell et dont l'exécution engendre la création d'un processus fils du processus shell. Un shell permet également d'exécuter des fichiers, appelés des scripts, contenant une suite de commandes plus ou moins complexes. Il offre aussi la possibilité à l'utilisateur de configurer son environnement de travail.

Il existe un certain nombre de shells que l'on peut séparer en deux familles :

La famille des Bourne shell :

- sh : le plus ancien développé chez AT&T par Steven Bourne fin 1970,
- ksh : le korn shell développé chez AT&T par David Korn en 1980, compatible avec sh et disposant de plus de fonctionnalités,
- bash : le bourne again shell développé par la Free Software Foundation en 1989,
- zsh : un des derniers nés (1990) de la famille, très complet.

La famille des C-shell (Université de Berkeley) :

- csh : dont le langage de programmation est très proche du langage C,
- tcsh : compatible avec le C-shell et beaucoup plus complet.

On appelle login-shell, le shell attribué (par défaut) à un utilisateur à l'issue de la procédure de login. À son lancement, ce shell exécute des commandes lues dans des fichiers de configuration.

Pour changer temporairement de shell en cours de session :

`nom_du_shell` ↩

`exit` ou `ctrl-d` pour quitter ce shell

4.2. Interprétation des commandes par le shell

3 étapes :

1. Lecture de la commande et séparation de celle-ci en mots

2. Application de mécanismes de substitution
3. Exécution : sans création de processus s'il s'agit d'une commande interne au shell, avec création de processus si c'est une commande externe (dans ce cas le fichier exécutable portant le nom de la commande est recherché, pour être exécuté, dans certains répertoires).

4.2.2. Les mécanismes de substitution

1. HISTORIQUE : Réappeler des commandes précédemment exécutées par le shell courant.
2. ALIAS : Attribuer des surnoms (alias) à des commandes particulières.
3. VARIABLES : Définir (ou activer) des variables pour paramétrer l'environnement ou pour écrire des scripts.

Un identificateur de variable est une chaîne de caractères alphanumériques (commençant par une lettre). Il existe deux types de variables :

- les variables d'environnement prédéfinies permettant de réaliser des contrôles particuliers : valeur du prompt, listes de répertoires, numéro du dernier processus lancé en arrière-plan. Certaines sont entièrement gérées par le shell, d'autres peuvent voir leur valeur modifiée par l'utilisateur.
- Les variables définies par l'utilisateur.

Parmi les variables d'environnement, on distingue deux catégories de variables :

➔ Les variables exportées dont les valeurs sont transmises aux applications lancées depuis le shell.

➔ Les variables locales dont les valeurs ne sont pas transmises aux applications lancées depuis le shell.

Pour obtenir la valeur d'une variable, on utilise le caractère spécial $: `$var` ou `${var}`.

4. EXPANSION DE FICHIERS : Utiliser des caractères spéciaux pour spécifier des listes de fichiers.

Caractères	Signification
*	Chaîne de caractères quelconque (éventuellement vide)
?	Caractère unique quelconque
[	Début de définition d'un ensemble de caractères
[!	Début de définition du complémentaire d'un ensemble de caractères
]	Fin de définition d'un ensemble de caractères
-	Marque d'intervalle dans un ensemble

4.2.2. Les caractères spéciaux communs aux différents shells

Il existe un ensemble de caractères interprétés par le shell d'une manière particulière. Pour qu'un caractère spécial soit traité comme un caractère quelconque il faut le faire précéder du caractère \.

Remarque : Si le caractère NEWLINE est précédé du caractère \, il est remplacé par le caractère ESPACE.

Caractères	Signification
\	Despécialisation du caractère suivant
ESPACE et TAB	Séparateurs de mots
NEWLINE	Fin de commande
&	Lancement d'une commande en arrière-plan
|	Tube
;	Séparateur de commandes
< ≪ ≫ >	Redirection des E/S standards
( )	Délimiteurs de commandes
' ` "	Délimiteurs de chaînes
`$var`	Valeur de la variable `var`
#	Début d'un commentaire qui va jusqu'à la fin de la ligne dans un fichier interprété par shell
&&	Exécution conditionnelle de commandes : `com1 && com2` ⇔ `com1` est exécutée et si `com1` aboutit, `com2` est exécutée
||	Exécution conditionnelle de commandes : `com1 || com2` ⇔ `com1` est exécutée et si `com1` échoue, `com2` est exécutée
%	Référence à une tâche dans le mécanisme de job control
∼`nom_logique`	Symbolise le répertoire privé d'un utilisateur

Les délimiteurs de chaînes de caractères :

- les quotes simples : Dans une chaîne délimitée par des quotes simples, tous les caractères perdent leur aspect spécial à l'exception du NEWLINE et du !. La seule substitution réalisée est donc celle liée à l'historique.

- les anti-quotes : Une chaîne délimitée par des anti-quotes est interprétée comme une commande et le résultat de la commande lui est substitué.

- les quillemets : Dans une chaîne délimitée par des guillemets, les caractères NEWLINE, !, $, \ et ` sont les seules à être interprétés. Il faut noter que le mécanisme de substitution lié aux variables ne peut pas être inhibé (même si $ est précédé de \).

4.3. Le C-shell

Lancement : **csh**

Terminaison : **ctrl-d** ou **logout** ou **exit**

4.3.3. Historique

À chaque fois que l'utilisateur soumet une commande, la commande est enregistrée par le shell (avant l'application des mécanismes de substitution autre que celui lié à l'historique). Il est possible de rappeler une commande précédente (ou une partie de celle-ci) telle quelle ou après lui avoir fait subir quelques modifications.

`history` : commande interne au C-shell pour afficher la liste des commandes mémorisées.

`!` : déclenche le mécanisme de substitution lié à l'historique, sauf lorsqu'il est suivi d'un caractère d'espacement - espace, tabulation, newline - ou des signes = ou (.

Une référence à l'historique peut être composée de une à trois parties dans l'ordre suivant :

1. une référence à une ligne de commande (event designator),
2. une référence à un mot dans une ligne de commande (word designator),
3. un ou plusieurs modifieur(s) de mot(s) référencés.

Event Designator :

Designateur	Substitué par
!!	La commande précédente
!n	La ligne de commande de numéro n
!-n	La ligne de commande de numéro : numéro de la commande courante - n
`!str`	La ligne de commande la plus récente commençant par la chaîne `str`
`!?str`	La ligne de commande la plus récente contenant la chaîne `str`

Word Designator :

Designateur	Substitué par
:0	Le premier mot = nom de la commande
:n	Le nième mot
^	Le deuxième mot (⇔ :1)
$	Le dernier mot
:n-m	Tous les mots allant du nième au mième
*	Tous les arguments
:n*	Tous les mots à partir du nième

Modifieurs dédiés au traitement des noms de fichier :

:h	Supprime la partie droite d'une référence absolue à un fichier (reste le chemin d'accès)
:t	Supprime le chemin d'accès dans une référence absolue à un fichier (reste le nom du fichier)
:r	Supprime le suffixe le plus à droite d'un nom de fichier
:e	Ne conserve que le suffixe d'un nom de fichier

Modifieurs permettant d'effectuer des substitutions :

`:s/str1/str2[/]`	La 1ère occurence de la chaîne `str1` dans la chaîne référencée est remplacée par la chaîne `str2`. Le caractère `/` final peut être omis lorsque `str2` est immédiatement suivi du caractère newline
`&`	Répète sur la chaîne référencée la dernière substitution effectuée à l'aide d'un modifieur de la forme `:s/str1/str2[/]`. Cette substitution peut appartenir à la ligne de commande courante (mais dans la partie située à gauche du `&`.
`:gs/str1/str2[/]` ou `g&`	Permet d'appliquer la substitution à chacun des mots de la chaîne référencée

4.3.3. Alias

Définition d'un alias :

alias `nom_alias commande`

alias ↩ : donne la liste des alias définis

Suppression d'un alias :

unalias `nom_alias`

Pour manipuler les paramètres des alias :

`\! :n` = désigne le nième paramètre
`\!$` = désigne le dernier paramètre
`\!^` = désigne le premier paramètre
`\!*` = désigne la liste des paramètres

Exemples :

```
%alias ll ls -l \!:2
%alias
ll      (ls -l !=2)
%ll p1.0 p2.c p3.c p4.c
-rw------- 1 vg ens 7 nov 10 19:05 p2.c
%alias cd 'cd \!:1; pwd; echo il y a 'ls | wc -l'fichiers'
```

Remarque : Les alias définis dans un processus C-shell ne sont pas exportés.

4.3.3. Les variables du C-shell

Commandes de manipulation des variables

set `var` : définit la variable locale `var` (= la chaîne vide).

unset `var` : supprime la définition de la variable locale `var`.

set `var=chaine` : affecte à la variable locale `var` la valeur `chaine`.

set : affiche la liste des valeurs des variables locales.

setenv `var` : définit la variable d'environnement `var`.

unsetenv `var` : supprime la définition de la variable d'environnement `var`.

setenv `var chaine` : affecte à la variable d'environnement `var` la valeur `chaine`.

printenv : affiche la liste des valeurs des variables d'environnement. Particularités : permet de manipuler des variables de type tableau et d'évaluer des expressions arithmétiques.

Le mécanisme de substitution de variables

Pour afficher la valeur d'une variable :

echo `$var`

`$ var`	La valeur de la variable est substituée
`$var[indice]`	où `var` est de type tableau et, `indice` est, soit un entier, soit un intervalle de la forme `[entier1,entier2]`, soit * pour tous les indices. Les valeurs des mots indicées par `indice` dans `var` sont substituées
`$#var`	Le nombre de mots dans `var` est substitué
`$?var`	substitue 1 ou 0 selon que `var` est définie ou non
`$0`	le nom du fichier en cours d'exécution est substitué

Les variables prédéfinies du C-shell

Les variables locales :

Nom	Interprétation
`argv`	Liste des arguments du shell
`cdpath`	Liste des répertoires utilisés par la commande `cd` pour rechercher le répertoire donné par les paramètres
`cwd`	Référence absolue du répertoire de travail
`home`	Référence absolue du répertoire privé
`path`	Liste des répertoires de recherche des commandes
`prompt`	Chaîne de caractères utilisée comme invité (par défaut %)
`shell`	Référence absolue du shell
`status`	Code de retour de la dernière commande exécutée
`term`	Type de terminal utilisé

Les variables d'environnement :

Nom	Interprétation
`HOME`	Valeur déduite de la variable home
`LOGNAME`	Nom de l'utilisateur
`PATH`	Valeur déduite de la variable `path`
`SHELL`	Valeur déduite de la variable `shell`
`TERM`	Valeur déduite de la variable `term`

Les variables qui, si elles sont définies, modifient le comportement du C-shell :

Nom	Effet
`echo`	Echo des commandes après réalisation des substitutions et avant leur exécution
`filec`	Activation du mécanisme de complétion des noms de fichiers
`history`	Activation du mécanisme d'historique. Sa valeur définit le nombre de lignes de commandes mémorisées
`ignoreeof`	Ignorer le caractère `eof` pour terminer le processus c-shell associé
`noclobber`	Redirection > sur un fichier existant ou ≫ sur un fichier non existant impossible
`noglob`	Occultation du mécanisme d'expansion des noms de fichiers
`notify`	Envoi de messages lors d'un changement d'état d'une tâche dans le processus de job control
`savehist`	Sauvegarde en fin de shell (dans le fichier `$HOME/.history`) d'un certain nombre de lignes de commandes qui seront automatiquement rechargées au lancement suivant
`verbose`	Affichage de chaque commande après substitution de l'historique

Les expressions arithmétiques

Una variable peut recevoir la valeur d'une expression par une commande de la forme :

```
@var opérateur expr
```

`opérateur` = opérateur d'affectation du C (`= += -= *= /= %=`)

4.3.3. Fichiers de configuration du C-shell

Un fichier de configuration est un fichier de commandes shell dont le nom et l'emplacement dans le système de fichiers sont prédéfinies et qui permettent à l'utilisateur ou à l'administrateur de personnaliser son environnement. Il existe en C-shell trois fichiers de configuration (placés à la racine du répertoire privé de l'utilisateur) :

`.cshrc` : interprété au démarrage de tout processus C-shell

`.login` (facultatif) : interprété au démarrage du C-shell de login à la suite du `.cshrc`

`.logout` : interprété à la terminaison (avec exit ou logout) du C-shell de login.

Exemple de fichier `.cshrc`

```
# @(#) cshrc 1.11 89/11/29 SMI
umask 022
set history=32
set prompt = "%"

setenv OPENWINHOME /usr/openwin
setenv XNEWSHOME $OPENWINHOME
```

```
set path = (. ~ ~/bin /usr/local/bin )
alias ll ls -al
alias emacs emacs -nolisp
#Les alias définis dans le .cshrc sont donc
connus par tous les csh exécutés
```

4.3.3. Gestion de processus par le mécanisme de job control

Le job control est un système de gestion de processus pris en charge par l'interprêteur de commandes.

Chaque commande interprétée par un shell correspond à un job ou tâche. Un job peut correspondre à plusieurs processus dans le cas où la commande est un lancement concurrent de processus communiquant par pipes. Toute tâche possède une identification interne au shell et indépendante du PID.

Exemple :

```
%gcc infini.c -o infini
%infini &
[1] 360
```

Le processus associé à l'exécution du programme `infini` s'exécute en arrière-plan. L'interpréteur shell répond en indiquant entre crochets le numéro de la tâche (ici [1]) représentant l'exécution de la commande et, le PID du processus créé (ici 360).

Les tâches gérées par un shell peuvent se trouver dans l'un des trois suivants :

- En premier plan : Une tâche en premier plan correspond à un groupe de processus qui lisent et écrivent sur le terminal de lancement. Si aucune tâche n'est en premier plan, c'est le processus shell qui lit au terminal. Évidemment au plus une tâche peut être au premier plan. Par exemple,
  `%gcc infini.c -o infini`
  cette commande est une tâche lancée en premier plan.
- En arrière plan : Les processus appartenant à une tâche en arrière plan ne peuvent pas lire sur le terminal et, éventuellement, ne peuvent pas y écrire. Par exemple,
  `%gcc infini.c -o infini &`
  cette commande est une tâche lancée en arrière plan.
- Stoppé ou suspendu : Les processus appartenant à une tâche stoppée sont en sommeil et attendent que l'utilisateur demande le passage à l'un des deux états précédents.

Pour désigner une tâche, on peut utiliser :

`%num`	tâche de numéro num
`%chaine`	tâche dont le nom de commande commence par `chaine`
`%?chaine`	tâche dont le nom de commande contient `chaine`

Pour gérer ses tâches, l'utilisateur peut utiliser les commandes internes au c-shell suivantes :

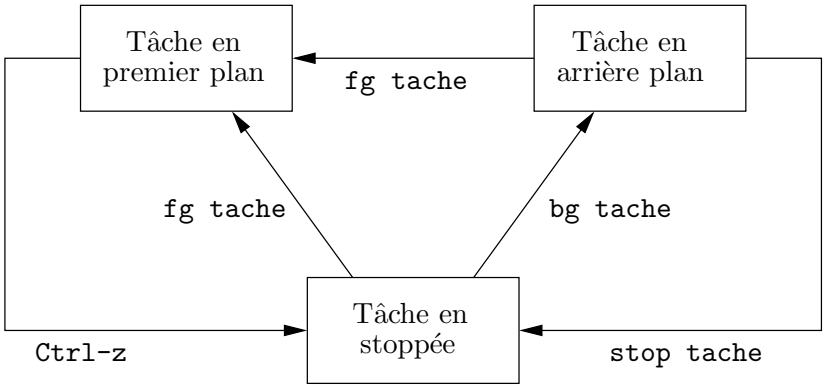
`jobs [-1]`	liste les tâches gérées par le chell.

Exemple :

```
%jobs
 [1]   + Running       infini
%jobs -l
 [1]   +    360 Running      infini
%ps
 PID     S    TT       COMMAND
 360     R    pts/3    infini
```

`kill [-n] tache...` adresse le signal n aux tâches spécifiées en argument.

Les commandes permettant de faire passer une tâche d'un état à un autre sont présentées dans le schéma ci-dessous :



Exemple :

```
{La tâche de numéro 1 est en arrière plan}
%stop %1
{La tâche de numéro 1 est suspendue}
%jobs
 [1]   + Stopped (signal)   infini
%fg %1
infini
{L'exécution de la tâche de numéro 1 est reprise en premier plan}
^Z
Stopped (user)
{L'utilisateur a adressé à la tâche en premier plan un signal d'interruption - en
tapant ctrl-z au clavier}
%jobs
 [1]   + infini &
{L'exécution de la tâche de numéro 1 est reprise en arrière plan}
%jobs
 [1]   + Running             infini
%kill %1
%jobs
 [1]   + Terminated          infini
```

Remarque : Le fait de terminer un processus c-shell provoque l'envoi aux tâches gérées par le c-shell d'un signal d'interruption. Les tâches en arrière plan sont protégées contre ce signal en c-shell qui continuent donc à être exécutées.

4.3.3. Les scripts en C-shell

Exécution de fichiers de commandes

`fich` = fichier contenant des commandes C-shell

`csh fich`	exécuter le fichier ordinaire `fich` par un processus csh fils,
`fich`	exécuter le fichier exécutable `fich` par un processus csh fils si la première ligne de `fich` commence par `#` ou par `#!réf-abs-csh`
`source fich`	exécute le fichier ordinaire `fich` dans le shell courant

Liste des paramètres d'une commande

`argv` : variable de type tableau de chaînes de caractères contenant la liste des paramètres d'une commande

`$argv` ou `$*`	tous les paramètres
`$argv[n]` ou `$n`	valeur du nième paramètre
`$#argv`	nombre de paramètres

Structures de contrôle

Structure conditionnelle

```
if (expression) commande
if (expression)
      then suite-de-commandes
      else suite-de-commandes
endif
switch (valeur)
      case valeur1 :
           suite-de-commandes
           breaksw
      ...
      case valeuri :
           suite-de-commandes
           breaksw
      ...
      default :
           suite-de-commandes
endsw
```

Sturcture répétitive

```
repeat n commande
```

= exécute `commande` n fois

```
foreach var(liste-de-valeurs)
      suite-de-commandes
end
```

= exécute itérativement `suite-de-commandes` en affectant successivement à la variable `var` les valeurs listées entre parenthèses

```
while(expression)
      suite-de-commandes
end
```

Étiquette et branchement

```
etiquette : commande
goto etiquette
```

Sortie de boucle

```
break
```

= sortie de la boucle la plus interne incluant l'instruction `break`

Expression booléenne

Soit comparaison entre valeurs numériques (<,<=, >, >=, ==, =, &&, ||), soit requète sur le statut d'un fichier

Requête	Retourne vrai si
`-r fich`	L'utilisateur a la permission de lire `fich`
`-w fich`	L'utilisateur a la permission d'écrire sur `fich`
`-x fich`	L'utilisateur a la permission d'exécuter `fich`
`-e fich`	`fich` existe
`-o fich`	L'utilisateur est propriétaire de `fich`
`-z fich`	`fich` est vide
`-f fich`	`fich` est un fichier ordinaire
`-d fich`	`fich` est un répertoire

4.4. Le Bourne-Again-Shell

Les principales différences entre le c-shell (csh) et le bourne-again-shell (bash) sont présentées ici.

Pour les fichiers de configuration lus lors d'un login shell, il en existe principalement 3 pour le bourne-again-shell. Le premier fichier interprété est le fichier `/etc/profile` qui est donc un fichier de configuration défini par l'administrateur système. Les deux fichiers suivants de nom .bashrc et .profile doivent être placés à la racine du répertoire privé de l'utilisateur auquel ils permettent de personaliser son environnement shell à sa convenance.

Les caractères spéciaux qui ont des spécificités différentes en c-shell et en bourne-again-shell sont les suivants :

Caractères	Spécificité bourne-again-shell
!	Introduit une référence à l'historique
~	Remplacé par le nom du répertoire privé de l'utilisateur, quand il est placé au début d'un mot
{ et }	Utilisés pour la substitution de noms de fichier
^	Utilisé dans les substitutions basées sur l'historique
%	Référence à une tâche (ou job) dans le mécanisme de job control
#	Introduction d'un commentaire uniquement dans un fichier de commandes

En Bourne-Again-Shell, pour la séparation des commandes simples en mots, les caractères considérés comme séparateurs sont les caractères de la chaîne contenue dans la variable IFS ; l'utilisateur a donc la possibilité de modifier ce comportement, cependant les caractères par défaut sont ESPACE et TABULATION. De plus, certains caractères sont considérés comme des mots en eux-même, qu'ils soient ou non entourés de caractères séparateurs. Ces caractères sont donc eux aussi des caractères séparateurs de mots, ce sont des caractères séparateurs de commandes simples ainsi que les caractères de redirections ¡ et ¿ et les parenthèses (et).

Attention, en bourne-again-shell, la complétion de noms de fichiers n'est pas effectuée sur les mots qui doivent être interprétés comme des noms de fichiers sur lesquels porte une redirection. Par contre, les mécanismes de substitution de variables et de commandes sont appliqués à ces mots de la même manière qu'en c-shell. Dans le domaine des redirections, le bourne-again-shell offre des fonctionalités plus étendues que le c-shell. Alors que le c-shell ne permet de rediriger que 3 flux d'entrées sorties standards, le bourne-again-shell autorise lui la redirection de n'importe quel flux d'entrées sorties, à partir du moment que l'on en connaît le numéro.

4.4.4. Découpage des chemins

Les scripts shell manipulent souvent chemins (pathnames) et noms de fichiers. Les commandes basename et dirname sont très commodes pour découper un chemin en deux partie (répertoires, nom de fichier) :

```
$ dirname /un/long/chemin/vers/toto.txt
```

```
/un/long/chemin/vers
$ basename /un/long/chemin/vers/toto.txt
toto.txt
```

4.4.4. Évaluation de commandes

Il est courant de stocker le résultat d'une commande dans une variable. Nous entendons ici par "résultat" la chaîne affichée par la commande, et non son code de retour. Bash utilise plusieurs notations pour cela : les back quotes (‘) ou les parenthèses :

```
$ REP=‘dirname /un/long/chemin/vers/toto.txt‘
$ echo $REP
/un/long/chemin/vers
```

ou, de manière équivalente :

```
$ REP=$(dirname /un/long/chemin/vers/toto.txt)
$ echo $REP
/un/long/chemin/vers
```

(attention encore une fois, pas d'espaces autour du signe égal). La commande peut être compliquée, par exemple avec un tube :

```
$ Fichiers=$(ls /usr/include/*.h | grep std)
$ echo $Fichiers
/usr/include/stdint.h /usr/include/stdio_ext.h /usr/include/stdio.h
/usr/include/stdlib.h /usr/include/unistd.h
```

4.4.4. Découpage de chaînes

Bash possède de nombreuses fonctionnalités pour découper des chaînes de caractères. L'une des plus pratiques est basée sur des motifs. La notation ## permet d'éliminer la plus longue chaîne en correspondance avec le motif :

```
$ Var='tonari no totoro'
$ echo ${Var##*to}
ro
```

ici le motif est *to, et la plus longue correspondance "tonari no toto"1. Cette forme est utile pour récupérer l'extension (suffixe) d'un nom de fichier :

```
$ F='rep/bidule.tgz'
$ echo ${F##*.}
tgz
```

La notation # (un seul #) est similaire mais élimine la plus courte chaîne en correspondance :

```
$ Var='tonari no totoro'
$ echo ${Var#*to}
nari no totoro
```

De façon similaire, on peut éliminer la fin d'une chaîne :

```
$ Var='tonari no totoro'
$ echo ${Var%no*}
tonari
```

Ce qui permet de supprimer l'extension d'un nom de fichier :

```
$ F='rep/bidule.tgz'
$ echo ${F%.*}
rep/bidule
```

% prend la plus courte correspondance, et %% prend la plus longue :

```
$ Y='archive.tar.gz'
$ echo ${Y%.*}
archive.tar
$ echo ${Y%%.*}
archive
```

4.4.4. Opérateurs de comparaison

Le shell étant souvent utilisé pour manipuler des fichiers, il offre plusieurs opérateurs permettant de vérifier diverses conditions sur ceux-ci : existence, dates, droits. D'autres opérateurs permettent de tester des valeurs, chaînes ou numériques. Le tableau ci-dessous donne un aperçu des principaux opérateurs :

Opérateur	Description	Exemple
-e filename	vrai si filename existe	[-e /etc/shadow]
-d filename	vrai si filename est un répertoire	[-d /tmp/trash]
-f filename	vrai si filename est un fichier ordinaire	[-f /tmp/glop]
-L filename	vrai si filename est un lien symbolique	[-L /home]
-r filename	vrai si filename est lisible (r)	[-r /boot/vmlinuz]
-w filename	vrai si filename est modifiable (w)	[-w /var/log]
-x filename	vrai si filename est exécutable (x)	[-x /sbin/halt]
file1 -nt file2	vrai si file1 plus récent que file2	[/tmp/foo -nt /tmp/bar]
file1 -ot file2	vrai si file1 plus ancien que file2	[/tmp/foo -ot /tmp/bar]
-z chaine	vrai si la chaine est vide	[-z "$VAR"]
-n chaine	vrai si la chaine est non vide	[-n "$VAR"]
chaine1 = chaine2	vrai si les deux chaînes sont égales	["$VAR" = "totoro"]
chaine1 != chaine2	vrai si les deux chaînes sont différentes	["$VAR" != "tonari"]
num1 -eq num2	égalité	[$nombre -eq 27]
num1 -ne num2	inégalité	[$nombre -ne 27]
num1 -lt num2	inférieur (¡)	[$nombre -lt 27]
num1 -le num2	inférieur ou égal (¡ =)	[$nombre -le 27]
num1 -gt num2	supérieur (¿)	[$nombre -gt 27]
num1 -ge num2	supérieur ou égal (¿ =)	[$nombre -ge 27]

Quelques points délicats doivent être soulignés :

– Toutes les variables sont de type chaîne de caractères. La valeur est juste convertie en nombre pour les opérateurs de conversion numérique.

– Il est nécessaire d'entourer les variables de guillemets (") dans les comparaisons. Le code suivant affiche "OK" si $var est égale à "tonari no totoro" :

```
if [ "$myvar" = "tonari no totoro" ]
then
   echo "OK"
fi
```

Par contre, si on écrit la comparaison comme if [$myvar = "tonari no totoro"] le shell déclenche une erreur si $myvar contient plusieurs mots. En effet, la substitution des variables a lieu avant l'interprétation de la condition.

4.4.4. Scripts Bourne-Again-Shell

Concernant les scripts en bash, il n'y a qu'un seul espace d'allocation des variables et qu'une seule syntaxe pour l'affectation des variables : $< nom_var >= [valeur]$ (attention à ne pas mettre d'espace de part et d'autre du signe =). Néanmoins, comme en c-shell, la distinction entre variables locales et variables d'environnement existe. Mais contrairement au c-shell, une même variable peut passer du statut de variable locale à celui de variable d'environnement à l'aide de la commande `export`.

Arguments de la ligne de commande Lorsqu'on entre une commande dans un shell, ce dernier sépare le nom de la commande (fichier exécutable ou commande interne au shell) des arguments (tout ce qui suit le nom de la commande, séparés par un ou plusieurs espaces). Les programmes peuvent utiliser les arguments (options, noms de fichiers à traiter, etc).

En bash, les arguments de la ligne de commande sont stockés dans des variables spéciales :

```
$0, $1, ...  les arguments
$#  le nombre d'arguments
$*  tous les arguments
```

Le programme suivant illustre l'utilisation de ces variables :

```
#!/bin/bash

echo 'programme  :' $0
echo 'argument 1 :' $1
echo 'argument 2 :' $2
echo 'argument 3 :' $3
echo 'argument 4 :' $4
echo "nombre d'arguments :" $#
echo "tous:" $*
```

Exemple d'utilisation, si le script s'appelle "myargs.sh" :

```
$ ./myargs.sh un deux trois
programme  : ./myargs.sh
argument 1 : un
argument 2 : deux
```

```
argument 3 : trois
argument 4 :
nombre d'arguments : 3
tous: un deux trois
```

Autres structures de contrôle Instruction if :

```
#!/bin/bash

if [ "${1##*.}" = "tar" ]
then
        echo $1 est une archive tar
else
        echo $1 n\'est pas une archive tar
fi
```

Ce script utilise la variable $1, qui est le premier argument passé sur la ligne de commande.

Boucle for : comme dans d'autre langages (par exemple python), la boucle for permet d'exécuter une suite d'instructions avec une variable parcourant une suite de valeurs. Exemple :

```
for x in un deux trois quatre
do
    echo x= $x
done
```

affichera :

```
x= un
x= deux
x= trois
x= quatre
```

On utilise fréquemment for pour énumérer des noms de fichiers, comme dans cet exemple :

```
for fichier in /etc/rc*
do
   if [ -d "$fichier" ]
   then
       echo "$fichier (repertoire)"
   else
       echo "$fichier"
   fi
done
```

Ou encore, pour traiter les arguments passés sur la ligne de commande :

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

```
for arg in $*
do
  echo $arg
done
```

Boucle while : la même syntaxe est utilisée que pour la boucle for :

```
while commande
do
   liste-de-commandes
done
```

Instruction case : l'instruction case permet de choisir une suite d'instruction suivant la valeur d'une expression :

```
case "$x" in
  go)
      echo "demarrage"
      ;;
  stop)
      echo "arret"
      ;;
   *)
      echo "valeur invalide de x ($x)''
esac
```

Noter les deux ; pour signaler la fin de chaque séquence d'instructions.

Définition de fonctions Il est souvent utile de définir des fonctions. La syntaxe est simple :

```
mafonction() {
   echo "appel de mafonction..."
}

mafonction
mafonction
```

qui donne :

```
appel de mafonction...
appel de mafonction...
```

Voici pour terminer un exemple de fonction plus intéressant :

```
tarview() {
    echo -n "Affichage du contenu de l'archive $1 "
    case "${1##*.}" in
        tar)
           echo "(tar compresse)"
```

```
            tar tvf $1
         ;;
         tgz)
            echo "(tar compresse gzip)"
            tar tzvf $1
         ;;
         bz2)
            echo "(tar compresse bzip2)"
            cat $1 | bzip2 -d | tar tvf -
         ;;
         *)
            echo "Erreur, ce n'est pas une archive"
         ;;
    esac
}
```

Plusieurs points sont à noter :
- echo -n permet d'éviter le passage à la ligne ;
- La fonction s'appelle avec un argument ($1) : tarview toto.tar

Références bibliographiques :

J-P Armspach, P. Colin, F. Ostré-Waerzeggers, Unix Initiation et utilisation, *Dunod*, 1999.
J.-M. Rifflet, La programmation sous Unix, *Ediscience*,1999.