

Université de Nice Sophia-Antipolis

Unix

et

programmation du shell

Version 3.0

Richard Grin

31 juillet 1998

Table des matières

Présentation du polycopié

I Connaissances de base

1 Généralités sur Unix

1.1	Systèmes d'exploitation	1
1.2	Historique	1
1.3	Standards et versions d'Unix	1
1.4	Propriétés d'Unix	1
1.5	Structure d'Unix	1
1.5.1	Noyau	1
1.5.2	Shell	1
1.6	Organisation des systèmes de fichiers	1
1.6.1	Système de fichiers	1
1.6.2	Types de fichiers	1

2 Acteurs/objets du monde Unix

2.1	Utilisateurs	1
2.1.1	Un utilisateur dans le système	1
2.1.2	Super-utilisateurs	1
2.1.3	Administrateur du système	1
2.2	Processus	1
2.2.1	Signaux envoyés aux processus	1
2.2.2	Terminal de contrôle d'un processus, exécution en arrière-plan	1
2.2.3	Propriétaires et groupes effectifs et réels d'un processus	1
2.2.4	Création d'un nouveau processus, arbre des processus	1
2.3	Fichiers	1
2.3.1	i-node	1
2.3.2	Structure interne des répertoires	1
2.3.3	Propriétaire et groupe d'un fichier	1
2.3.4	Mode d'accès au fichier	1
2.4	Protection des fichiers	1

2.4.1	Droits associés aux autorisations	16
2.4.2	Autorisations “set user id” et “set group id”	16
2.4.3	Mécanisme de protection des fichiers	17
3	Premiers pas dans le système	18
3.1	Entrée dans le système	18
3.1.1	Changer son mot de passe (passwd)	18
3.1.2	Démarrage d’une session	19
3.2	Sortie du système	19
3.3	Touches spéciales	19
3.4	Format des commandes Unix	20
3.5	Consultation du manuel en ligne (man)	21
3.6	Nom d’un fichier, nom absolu, nom relatif	22
3.7	Commandes	22
3.7.1	Lancer une commande, supprimer un processus	22
3.7.2	Nom d’une commande, variable PATH	22
3.7.3	Nom complet et type d’une commande (whence, whereis)	23
3.7.4	Complétion des commandes par zsh	24
II	Commandes	25
4	Commandes liées à l’arborescence des fichiers	27
4.1	Visualisation de l’arborescence (ls)	27
4.2	Information détaillée sur les fichiers (ls -l)	28
4.2.1	Format d’affichage	28
4.2.2	Types de fichiers	29
4.2.3	Mode d’accès au fichier	29
4.2.4	Nombre de liens	30
4.3	Se déplacer dans l’arborescence (cd)	30
4.4	Afficher le répertoire courant (pwd)	30
4.5	Caractères spéciaux pour le shell	30
4.5.1	Génération des noms de fichiers (*, ?,)	30
4.5.2	Le caractère spécial ~	31
4.5.3	Le caractère spécial #	32
4.5.4	Autres caractères spéciaux	32
4.6	Afficher le type d’un fichier (file)	32
4.7	Rechercher des fichiers dans l’arborescence (find, locate)	32
5	Protection des fichiers	35
5.1	Changement des autorisations (chmod)	35
5.2	Masque pour les autorisations (umask)	36

6	Commandes d’observation du système	3
6.1	Date et Heure (date)	3
6.2	Nom de l’ordinateur (hostname)	3
6.3	Nom du système d’exploitation (uname)	3
6.4	Information sur les utilisateurs (who, finger)	3
6.5	Dernières connexions au système (last)	3
6.6	Système “ <i>Network Information Service</i> ” (NIS)	3
6.7	Espace disque occupé	3
6.7.1	Place occupée par la branche d’un répertoire (du)	3
6.7.2	Place libre d’un système de fichiers (df)	3
6.8	Nom du terminal utilisé (tty)	3
7	Commandes pour la gestion des processus	4
7.1	Processus en cours d’exécution (ps)	4
7.2	Supprimer un processus en cours d’exécution (kill)	4
7.3	Lancement automatique de processus à des moments donnés (at, crontab)	4
7.4	Gestion du plan d’un processus par le shell	4
8	Afficher, imprimer, envoyer le contenu d’un fichier	4
8.1	Afficher le contenu d’un fichier, concaténer plusieurs fichiers (cat) - Notion de redirection	4
8.2	Afficher les octets d’un fichier (od)	4
8.3	Afficher page à page (more) - Notion de pipe	4
8.4	Sorties sur les imprimantes	4
8.4.1	Informations sur le système d’impression (lpstat, printcap)	4
8.4.2	Lancement d’une requête d’impression (lpr)	4
8.4.3	Informations sur les requêtes d’impression (lpq)	4
8.4.4	Suppressions de requêtes d’impression (lprm, cancel)	4
8.5	Envoyer le contenu d’un fichier (mail)	4
8.6	Mise en page (pr)	4
8.7	Passer en Postscript (a2ps)	4
9	Gestion des fichiers	5
9.1	Copier des fichiers (cp)	5
9.2	Liens avec même numéro de i-node (ln)	5
9.3	Liens symboliques (ln -s)	5
9.4	Supprimer des fichiers (rm)	5
9.5	Déplacer, renommer des fichiers (mv)	5
9.6	Sauvegarder sur les lecteurs de disquettes des stations Sun	5
9.7	Compression et décompression (zip, gzip, compress)	5

10 Travail sur les répertoires	55
10.1 Créer un répertoire (mkdir)	55
10.2 Supprimer un répertoire (rmdir, rm -r)	55
10.3 Changer le nom d'un répertoire (mv)	55
10.4 Copier l'arborescence d'un répertoire (cp -r)	56
10.5 Lien symbolique (ln -s)	56
10.6 Pliage de répertoires en une seul fichier (tar, cpio)	57
11 Expressions régulières	58
11.1 Expressions régulières représentant un seul caractère	58
11.2 Expressions régulières représentant un ensemble de caractères d'un seul type	59
11.3 Autres expressions régulières	60
12 Éditeur de texte (emacs, xemacs)	62
12.1 Entrée et sortie	62
12.1.1 Lancer emacs	62
12.1.2 Sortir de emacs	63
12.2 Concepts de base	63
12.2.1 Description d'une fenêtre emacs	63
12.2.2 Commandes et associations de clés	64
12.2.3 Buffers et fenêtres	66
12.2.4 Point d'insertion, marque, régions	67
12.2.5 Mode de travail	68
12.3 Commandes	68
12.3.1 Commandes de base	68
12.3.2 Autres commandes	71
12.4 Personnalisation de emacs	71
12.5 Compléments pour les versions graphiques	72
13 Manipulation des données des fichiers	73
13.1 Tri (sort)	73
13.2 Recherche d'une chaîne de caractères (grep)	74
13.3 Compter les caractères, les mots, les lignes (wc)	74
13.4 Conversion, suppression de caractères (tr)	75
13.5 Fractionnement vertical (cut)	76
13.6 Comparaison du contenu de 2 fichiers	78
13.6.1 Différences entre deux fichiers texte (diff)	78
13.6.2 Égalité du contenu de 2 fichiers (cmp)	78
13.7 Traiter les lignes consécutives identiques (uniq)	79
13.8 Extraire le début ou la fin d'un fichier	79
13.8.1 Début d'un fichier (head)	79
13.8.2 Fin d'un fichier (tail)	79
13.9 Un éditeur non interactif (sed)	80

13.9.1 Description générale de l'éditeur	8
13.9.2 Structure d'une ligne de programme	8
13.9.3 Format pour indiquer les lignes à traiter	8
13.9.4 Exécution du programme	8
13.9.5 Commandes	8
13.10 Un langage d'édition de fichiers (awk)	8
13.10.1 Description générale de la commande	8
13.10.2 Champs	8
13.10.3 Structure d'un programme	8
13.10.4 Cas particuliers pour une ligne de programme	8
13.10.5 Sélecteurs	8
13.10.6 Actions	8
13.10.7 Fonctions, opérateurs	8
13.11 Exemples de programmes	8
13.12 Tableaux	8
13.13 D'autres commandes de manipulation de fichiers	8

III Mécanismes d'interprétation du shell

14 Interprétation du shell. Mécanismes de base	9
14.1 Généralités sur le shell	9
14.1.1 Définition	9
14.1.2 Compatibilité des différents shells	9
14.1.3 Commandes et processus, commandes internes au shell	9
14.1.4 Prompts	9
14.1.5 Environnement d'une commande ou d'un shellscript	9
14.1.6 Code retour d'une commande	9
14.2 Redirections	9
14.2.1 Redirections de la sortie standard	9
14.2.2 Redirection du fichier d'erreur (2>)	9
14.2.3 Envoi d'un message d'erreur (>&2)	9
14.2.4 Redirection de l'entrée standard (<)	9
14.2.5 Pseudo-fichier /dev/null	9
14.3 Pipe ()	9
14.4 Regroupement des commandes	9
14.4.1 Regroupement entre parenthèses	9
14.4.2 Regroupement entre accolades	9
14.5 Processus en arrière-plan (&)	9
14.5.1 Gestion des "jobs" par ksh ou zsh	9
14.6 Alias	9
14.7 Substitution de commande	9
14.8 Mécanismes d'interprétation	9

14.9	Inhiber l'interprétation du shell	99
14.10	Recherche d'une commande par le shell	100
14.11	Lancement de l'exécution d'un shellscrip	100
14.11.1	Lancement par le nom du shellscrip	100
14.11.2	Lancement par l'appel de la commande interne "."	102
14.11.3	Lancement par l'appel explicite d'un shell	102
14.12	Lancement explicite d'un shell	103
15	Variables, environnement	104
15.1	Paramètres, variables de position	104
15.1.1	Paramètres des shellscrips	104
15.1.2	Donner des valeurs aux paramètres de position (set)	104
15.2	Variables	105
15.2.1	Identificateur	105
15.2.2	Affectation	105
15.2.3	Désignation de la valeur de la variable (\$)	105
15.2.4	Décomposition en mots des valeurs des variables de zsh	106
15.2.5	Supprimer une variable (unset)	106
15.2.6	Variables spéciales du shell	106
15.2.7	Afficher la valeur d'une variable (echo)	108
15.2.8	Entrée de la valeur d'une variable au clavier (read)	108
15.2.9	Portée d'une variable (export), environnement de travail	109
15.2.10	Visualisation des variables disponibles (set, printenv)	110
15.3	Personnalisation de l'environnement	110
15.3.1	Options des shells (set, setopt)	110
15.3.2	Fichiers de personnalisation	110
15.4	Stratégie pour la personnalisation de l'environnement	112
16	Compléments sur le shell	113
16.1	Listes de commandes	113
16.2	Ordre de priorité	114
16.3	Fonctions	114
16.4	Compléments sur les redirections	115
16.4.1	Redirection de l'entrée standard sur fichier inclus (<<)	115
16.4.2	Redirection pour tout un shell (exec)	116
16.4.3	Descripteurs de fichier supérieurs à 2	116
16.4.4	Redirection vers un fichier désigné par son descripteur	117
16.4.5	Ordre d'évaluation des redirections	117
16.4.6	Redirection sans commande avec zsh	118
16.5	Compléments sur les variables	118
16.5.1	Valeurs par défaut pour les variables	118
16.5.2	Modifier l'environnement d'une commande	119
16.5.3	Facilités de ksh et zsh pour le traitement des valeurs de variables	119

16.5.4	Modification interactive de la valeur d'une variable sous zsh (vared)	12
16.5.5	Tableaux sous ksh et zsh	12
IV	Programmation	12
17	Programmation des shellscrips	12
17.1	Tests divers (test, [...], [[...]])	12
17.2	Décaler les paramètres de position (shift)	12
17.3	Sortie d'un shellscrip (exit)	12
17.4	Les structures de contrôle	12
17.4.1	if .. then .. else .. fi	12
17.4.2	case ... esac	12
17.4.3	for ... do ... done	12
17.4.4	while ... do ... done	13
17.4.5	until ... do ... done	13
17.4.6	Instructions liées aux boucles (continue, break)	13
17.4.7	Problèmes avec les boucles redirigées et les pipes	13
17.5	Interception des signaux	13
17.6	Commandes internes diverses	13
17.7	Récursivité dans l'exécution des shellscrips	13
17.8	Calculs, traitement des chaînes de caractères	13
17.8.1	Commande expr	13
17.8.2	Commande let	13
17.9	Traitement des chaînes de caractères	13
17.10	Aide pour traiter les options (getopts)	13
17.11	Mise au point des shellscrips (set -xv)	13

Présentation du polycopié

Ce polycopié est un support du cours donné dans différentes filières de l'Université de Nice-Sophia Antipolis.

Le cours est une introduction au système d'exploitation Unix. Le but est de présenter, à travers des cas concrets d'utilisation sous le système Unix, les concepts essentiels des systèmes d'exploitation utiles à un utilisateur averti.

La programmation des *shells* qui fait si souvent gagner un temps précieux dans l'exécution des tâches répétitives ou complexes est présentée à la fin de ce cours et en présente en fait l'aboutissement.

Les chapitres de ce support de cours ne suivent pas nécessairement l'ordre de leur présentation dans orale le cours. Ce polycopié est plutôt un manuel de référence qui aidera l'utilisateur à se remettre rapidement en mémoire les principales commandes et concepts d'Unix. L'exhaustivité (impossible de toute façon) n'a pas été recherchée. Une lecture du manuel en ligne (commande *man*) est indispensable pour connaître toutes les commandes et options de la dernière version installée. Mais cette aide en ligne est écrite en anglais et les informations essentielles sont souvent noyées dans les très nombreuses options et commandes, et les exemples sont trop souvent absents.

Ce cours n'aborde ni l'administration d'un système Unix ni la programmation système, même s'il donne quelques rudiments d'information concernant ces deux thèmes.

Il ne suppose aucune connaissance en informatique si ce n'est quelques notions élémentaires sur l'architecture des ordinateurs et sur la programmation.

Un autre polycopié rassemble les notions et commandes liées à l'utilisation des réseaux informatiques. Ces deux cours devront être complétés par quelques connaissances de base de l'interface graphique X Window, en particulier les notions de gestionnaire de fenêtres et de ressources.

Le lecteur intéressé par l'implémentation des différents Unix pourra consulter le livre "Unix Internals" écrit par Uresh Vahalia, éditions Prentice Hall (disponible à la bibliothèque de l'université de Nice).

Les remarques et les corrections d'erreurs peuvent être envoyées par courrier électronique à l'adresse grin@unice.fr, en précisant le sujet "Poly Unix et programmation du shell" et la date de la version du polycopié.

Première partie

Connaissances de base

Chapitre 1

Généralités sur Unix

1.1 Systèmes d'exploitation

Unix est un système d'exploitation.

Un système d'exploitation est un logiciel qui fournit un environnement d'exécution pour les programmes qui vont s'exécuter sur l'ordinateur. Il doit gérer en particulier les ressources que vont se partager les programmes. Il a, entre autres, la charge des fonctions suivantes, essentielles pour la bonne marche d'un ordinateur :

- gestion du processeur
- gestion de la mémoire centrale
- gestion du système de fichiers
- gestion des périphériques

Un ordinateur ne peut fonctionner sans système d'exploitation. Un ordinateur donné peut fonctionner avec plusieurs systèmes d'exploitation ; par exemple, certains PC peuvent mettre à l'utilisateur de travailler sous Unix ou sous Windows 95 ou NT.

1.2 Historique

1969 Ken Thompson écrit Unix en assembleur dans les laboratoires de Bell (AT&T).

1973 Dennis Ritchie et Ken Thompson réécrivent Unix en langage C (seulement 10 % en assembleur).

1974 Unix est distribué aux universités américaines.

1975 Première version d'Unix commercialisée (licence pour le code source ; version 6).

1977 Unix BSD (Berkeley Software Development) 1.0 de l'université de Berkeley aux États-Unis.

1979 Unix Version 7. Commercialisation d'une licence binaire (moins onéreuse que la licence pour le code source).

1984 Unix Système V

1991 Unix OSF/1

1.3 Standards et versions d'Unix

Unix est un système *ouvert* : il n'est pas lié à un constructeur d'ordinateur ou à une société d'édition de logiciels ; AT&T a largement diffusé le code source d'Unix dans les universités à ses débuts et de multiples Unix ont été développés par des sociétés différentes. Tous les grands constructeurs ont leur Unix, réécrit entièrement par eux ("Unix-like") ou développé à partir d'une licence achetée à AT&T ("Unix-based"). Tous ces Unix respectent un minimum de normes et il n'est pas trop difficile de porter une application d'un Unix vers l'autre.

Les différentes versions d'Unix étaient issues de Système V de AT&T ou de Unix BSD de l'université de Berkeley. Au début des années 1990, Open Software Fondation (OSF) a développé un nouvel Unix : OSF/1.

Le système *Solaris* actuellement utilisé sur de nombreux ordinateurs de l'université de Nice est issu de Unix Système V.

La plupart des commandes que vous utiliserez en tant qu'utilisateur ordinaire ou développeur d'applications sont les mêmes dans toutes les versions d'Unix. Les différences les plus sensibles se trouvent au niveau de l'administration du système.

La tendance actuelle est au rapprochement des différentes versions d'Unix, au moins au niveau de l'utilisateur et du développeur, autour de normes définies par des groupements d'utilisateurs, des constructeurs d'ordinateurs ou des éditeurs de logiciels. Ces normes spécifient des interfaces : elles définissent les signatures et sémantiques (mais pas les implémentations) des fonctions qui constituent ces interfaces. Les trois normes principales sont SVID (System V Interface Definition), POSIX (Portable Operating System based on UNIX) de l'IEEE et CAE (Common Applications Environment) de X/Open.

1.4 Propriétés d'Unix

Unix est un système d'exploitation

- d'usage général,
- multi-utilisateurs, multi-tâches,
- interactif,
- orienté temps partagé,

1.5. STRUCTURE D'UNIX

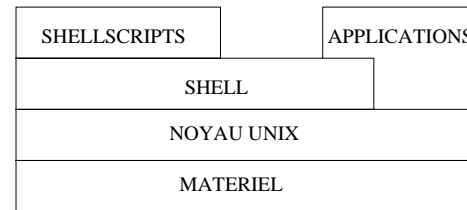


FIG. 1.1 – *Structure d'Unix*

- portable,
- dont les système de fichiers sont hiérarchisés en arbre,
- qui offre une compatibilité totale des entrées-sorties (pour Unix, les périphériques sont des fichiers ; voir 1.6.2),
- pour lequel il est très facile d'ajouter de nouvelles commandes sans modifier le noyau (grâce à la grande puissance des langages des shells).

1.5 Structure d'Unix

1.5.1 Noyau

Le noyau est la partie centrale d'Unix. Il se charge des tâches essentielles pour la bonne marche du système : gestion du système des fichiers, gestion du processeur et de la mémoire centrale. Les drivers de périphériques sont intégrés au noyau. Ils gèrent les échanges (les entrées-sorties) entre l'unité centrale et les périphériques.

Le noyau met à la disposition des autres programmes des procédures appelées *primitives*. Les autres programmes peuvent inclure dans leur code des *appels système* qui lancent l'exécution de ces primitives. Les primitives permettent de lancer de nouveaux processus, de lire ou d'écrire sur des fichiers, d'obtenir plus de place en mémoire centrale, etc... Ces appels système sont répertoriés dans la section 2 du manuel en ligne (voir 3.5).

Ce fonctionnement par primitives permet en particulier de résoudre les problèmes d'accès concurrent aux informations du système (sur un système mono-processeur). En effet, les appels système font entrer l'exécution en *mode noyau*. Dans ce mode le processus est assuré de garder le processeur jusqu'au retour au *mode utilisateur* quand l'appel système est terminé.

1.5.2 Shell

Le shell est l'interpréteur de commandes. Quand un utilisateur tape des commandes Unix, ces commandes sont lues par le shell qui effectue éventuellement des traitements.

avant de lancer l'exécution de la commande. Le shell est une couche logicielle bien séparée du noyau. Il existe plusieurs shells dont les plus utilisés sont :

- le *Bourne shell* *sh*, le shell standard d'Unix AT&T,
- le *C-shell* *csh*, le shell d'Unix BSD ; sa syntaxe rappelle le langage C,
- le *Korn-shell* *ksh* est une extension du Bourne shell. Il possède toutes les commandes du Bourne shell (et il se comporte presque exactement comme lui pour ces commandes) et il comprend aussi d'autres commandes et fonctionnalités qui facilitent le travail de l'utilisateur comme, par exemple, la gestion de l'historique des commandes tapées par l'utilisateur, qui existe aussi dans le C-shell. On le trouve maintenant dans la plupart des distributions Unix.
- le *Z shell* *zsh* est une extension du Korn-shell. Il offre en particulier la complétion et la correction de commande : l'utilisateur peut lui demander de compléter un nom de commande et il propose des corrections à l'utilisateur lorsque la commande comporte une erreur, par exemple lorsqu'une commande a été mal orthographiée.

Certaines commandes à la disposition de l'utilisateur, sont programmées dans le shell et celui-ci peut donc les exécuter directement. Elles sont peu nombreuses ; on trouve par exemple les commandes *cd* ou *pwd*. On les appelle les *commandes internes* au shell. Les autres commandes sont des *commandes externes* au shell. Pour les exécuter le shell lance un programme qui correspond à un fichier exécutable situé dans l'arborescence des fichiers.

Le shell possède un véritable langage avec des structures de programmation (alternatives, répétitions,...) et l'utilisateur peut écrire ses propres commandes dans ce langage (le programme s'appelle un *shellscript*). Une fois écrites, ces nouvelles commandes peuvent être utilisées exactement comme les commandes classiques d'Unix.

Le shell que nous utiliserons est *zsh*.

1.6 Organisation des systèmes de fichiers

1.6.1 Système de fichiers

Les fichiers d'Unix sont enregistrés dans un ensemble structuré hiérarchiquement en arbre, appelé système de fichiers. Un système de fichiers est composé d'une racine et de noeuds qui sont des *fichiers répertoires* (ces fichiers contiennent des références à d'autres fichiers), et de *fichiers ordinaires* qui contiennent des données et des programmes.

En général, plusieurs systèmes de fichiers sont "*montés*" sur le système "racine", c'est à dire celui qui contient le répertoire / (voir commande *df* en 6.7.2). Ces systèmes sont enregistrés sur des disques physiques différents ou sur un même disque mais sur des *partitions* différentes (un disque physique peut être divisé en plusieurs disques logiques appelés partitions). Tous ces systèmes de fichiers sont vus par l'utilisateur comme un seul système.

1.6. ORGANISATION DES SYSTÈMES DE FICHIERS

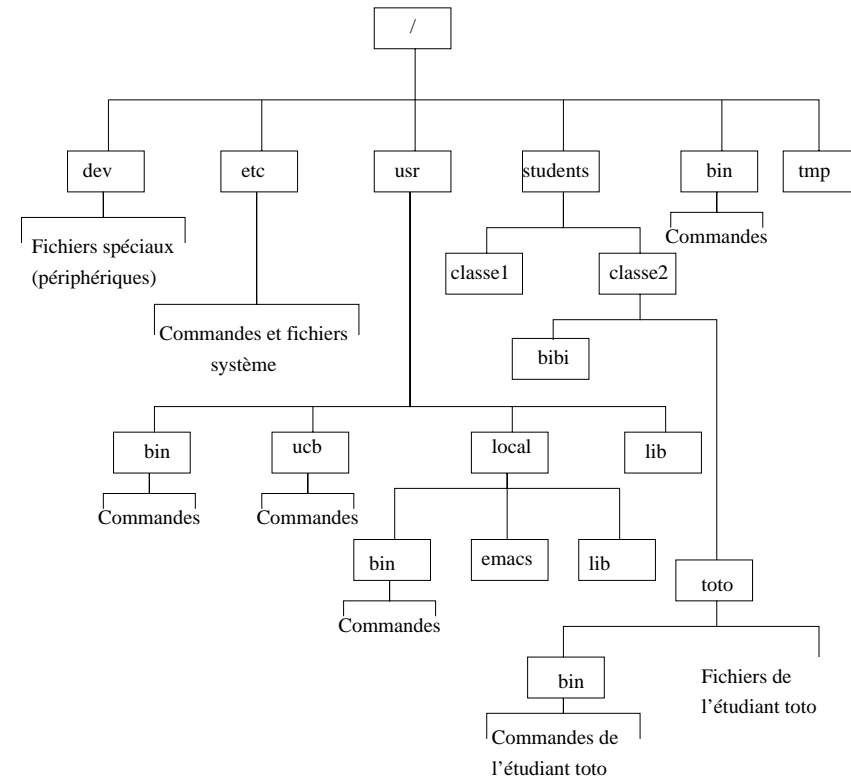


FIG. 1.2 – Arborescence des fichiers

Des utilitaires de réseaux (comme NFS) permettent même de voir des systèmes implantés sur des machines différentes comme s'ils appartenaient à une même arborescence.

L'arborescence totale ressemble à l'arborescence de la figure 1.2.

1.6.2 Types de fichiers

Unix utilise les types de fichiers suivants :

fichiers ordinaires : ils contiennent les données ou les programmes. Ils n'ont aucune structure particulière : ils sont considérés comme une suite d'octets. Il n'y a pas de notion d'enregistrement ni d'accès par index. Si on veut un séquentiel indexé, c'est le programme (ou on l'achète dans le commerce) !

On peut cependant distinguer trois grands types de fichiers ordinaires :

- les fichiers binaires exécutables, écrits dans le code du processeur de la machine, qui ont une structure particulière reconnue par le noyau Unix.
- les fichiers de textes qui sont structurés en lignes.

En Unix, le caractère de fin de ligne est par convention le caractère de code ASCII 10 (“*linefeed*” en anglais).

On remarquera que ce n’est pas le même caractère que celui utilisé par les systèmes d’exploitation MS-DOS (sur PC) ou Mac-OS (sur Macintosh), ce qui oblige à effectuer un traitement minimum pour transférer des fichiers de textes entre ces systèmes.

- les autres fichiers qui n’ont pas de structure particulière pour Unix (mais peuvent avoir une structure particulière adaptée au logiciel qui les a créés).

répertoires : ils sont les noeuds de la structure arborescente des fichiers. Ils contiennent les identificateurs (i-nodes) d’autres fichiers. Ils correspondent à des sous-bibliothèques ou dossiers qui contiennent d’autres fichiers.

fichiers spéciaux : ils sont liés à un périphérique (/dev/tty01 est par exemple lié à un terminal) ; ils correspondent à des programmes (pilotes ; *drivers* en anglais) qui gèrent les échanges avec les périphériques : disques, terminaux, imprimantes, streamers, lecteurs de bandes, etc.

liens symboliques : ils contiennent le nom d’un autre fichier et permettent des indirectes.

sockets : ils sont utilisés pour les liaisons inter-processus ; ils ne seront pas étudiés ici.

Unix Système V utilise aussi les “*pipes nommés*” qui sont utilisés pour la communication entre les processus. Ils ne seront pas étudiés ici.

Chapitre 2

Acteurs/objets du monde Unix

Ce chapitre expose quelques notions et concepts de base sur le fonctionnement d’Unix. Les commandes associées à ces notions seront étudiées dans les chapitres suivants.

Les acteurs/objets principaux que l’on rencontre en Unix sont les utilisateurs, les processus et les fichiers.

2.1 Utilisateurs

2.1.1 Un utilisateur dans le système

Nom, mot de passe, uid

Un utilisateur est repéré par son *nom* qu’il doit taper au moment de l’entrée dans le système.

Chaque nom correspond à un numéro d’utilisateur (un nombre entier) : le *uid* (*user identifier*). Plusieurs utilisateurs peuvent avoir un même uid mais cette possibilité n’est utilisée que pour quelques pseudo-utilisateurs du système qui ont par exemple besoin de autorisations du super-utilisateur (voir 2.1.2) pour accomplir une tâche bien définie.

Un utilisateur s’authentifie lors de son entrée dans le système grâce à son *mot de passe*.

Groupes d’un utilisateur

Un utilisateur appartient à un *groupe d’utilisateurs* dont le numéro est enregistré dans la ligne réservée à cet utilisateur dans le fichier `/etc/passwd` (ou dans les fichiers système qui jouent un rôle équivalents ; voir ci-dessous page 10 la section sur les fichiers système).

La notion de groupe permet à plusieurs utilisateurs de partager certains droits pour l’utilisation des fichiers appartenant au groupe (voir 2.4 et 2.3.3).

La plupart des Unix modernes permettent à un utilisateur d’appartenir à plusieurs groupes en plus de son groupe principal (celui qui est enregistré dans le fichier des mots de passe).

Les groupes ne sont pas gérés de la même façon par les différents types d'Unix. Nous n'étudierons pas cette notion en détails dans ce cours.

Répertoire HOME

Un utilisateur possède un *répertoire* “HOME” dans lequel il est positionné lorsqu'il entre dans le système. Ce répertoire est choisi par l'administrateur en fonction de l'utilisateur. En général, le nom terminal de ce répertoire est le nom de l'utilisateur et il est placé dans un répertoire réservé à sa fonction dans l'institution. Par exemple, l'étudiant “jean” en deuxième année d'informatique aura le répertoire HOME “/students/info2/jean”.

Programme de démarrage

L'administrateur attribue à chaque utilisateur un *programme de démarrage* qui sera exécuté quand l'utilisateur entrera dans le système. Pour vous, c'est le shell *zsh*.

Attributs d'un utilisateur

En résumé, à chaque utilisateur est associé :

- un nom,
- un mot de passe,
- un uid,
- un ou plusieurs groupes,
- un répertoire “HOME”,
- un programme de démarrage.

Fichiers système

Les informations sur un utilisateur sont enregistrées dans le fichier `/etc/passwd`. Pour des raisons de sécurité ou de commodité (pour gérer un réseau de machines), l'administrateur peut ajouter des utilitaires de gestion du système (“Yellow pages” ou NIS, “Kerberos”, etc.). Dans ce cas, le fichier `/etc/passwd` peut ne pas contenir toutes les informations concernant les utilisateurs (par exemple, les mots de passe sont remplacés par “*”) ou même aucune information sur certains utilisateurs (ces informations ne sont enregistrées que sur certaines machines du réseau ; voir en particulier 6.6).

2.1.2 Super-utilisateurs

Les utilisateurs de uid 0 sont appelés des super-utilisateurs. Ces utilisateurs ont des droits que les autres utilisateurs n'ont pas. Ils peuvent, par exemple, changer le mot de passe des autres utilisateurs et changer les autorisations ou le propriétaire de n'importe quel fichier de l'arborescence.

2.1.3 Administrateur du système

Un système Unix doit être géré par un administrateur qui est responsable du bon fonctionnement du système. Voici quelques unes de ses tâches :

- enregistrer les nouveaux utilisateurs et créer leur environnement de travail,
- installer les nouvelles versions des logiciels,
- repérer les problèmes qui empêchent le système de fonctionner correctement,
- nettoyer périodiquement le disque des fichiers non utilisés,
- effectuer régulièrement des sauvegardes des données et des programmes.

Pour beaucoup des tâches qu'il doit accomplir, l'administrateur doit entrer dans le système comme super-utilisateur.

2.2 Processus

L'unité d'exécution est le processus. Toute action est exécutée par un processus. Tous les Unix modernes offrent une unité plus légère appelée “thread” qui permet à un processus de faire exécuter une tâche par plusieurs “sous-tâches” qui s'exécutent en parallèle et partagent le même espace mémoire. Nous ne les étudierons pas dans ce cours.

Chaque processus est identifié par le système par un numéro de processus (*pid* ; *process identifier*).

Une commande tapée par l'utilisateur peut engendrer un ou plusieurs processus, ou n'engendrer aucun nouveau processus si la commande est interne au shell.

Unix est multi-tâche. Tous les processus en cours d'exécution se partagent le processeur de la machine. Chaque processus reçoit l'usage du processeur pendant un laps de temps assez court (environ un centième de seconde) pour que l'utilisateur ne s'aperçoive pas de la présence des autres processus pendant les périodes d'interaction avec le programme (entrées-sorties avec le clavier ou avec la souris).

Un processus travaille dans son propre environnement (en particulier avec ses propres variables). A un moment donné, son environnement est représenté par une *image* qui est une sorte de cliché (au sens photographique) du processus.

1. même si la machine est à plusieurs processeurs, ceux-ci sont habituellement bien moins nombreux que les processus

L'image d'un processus est composée de :

- son code
- les données associées (traitées par le code)
- les fichiers ouverts et leur état
- le répertoire courant
- les informations du système sur le processus (propriétaire, terminal associé, etc.)

On remarquera en particulier qu'à un moment donné de son exécution un processus a un *répertoire courant* où il est situé.

Tout le code du programme n'est pas nécessairement en mémoire durant l'exécution. Le code est découpé en *pages* par le système et seules les pages nécessaires sont chargées en mémoire centrale. Au cours de l'exécution du programme des pages peuvent être transportées du disque vers la mémoire centrale si la page n'est plus nécessaire et que le système a besoin de place en mémoire centrale, et faire le chemin inverse lorsque la page est nécessaire à l'exécution du programme. Ce système de gestion de la mémoire s'appelle le "*demand paging*". La zone du disque réservée à ces allers et retours s'appelle la zone de "*swap*".

2.2.1 Signaux envoyés aux processus

Durant son exécution, un processus peut recevoir un signal. La plupart des signaux interrompent l'exécution du processus mais celui-ci peut les intercepter. Il peut alors effectuer un traitement spécial et choisir ensuite de continuer à s'exécuter ou d'arrêter son exécution (voir 7.2 et 17.5).

Les signaux peuvent être envoyés par :

- un processus, avec la commande ou l'appel système `kill` (voir 7.2),
- le noyau, engendrés par la frappe d'une touche du clavier par l'utilisateur pour interrompre (ou stopper momentanément) tous les processus liés au terminal (notion étudiée en 2.2.2),
- le noyau, engendrés par des erreurs venant du matériel ou du logiciel.

Voici quelques numéros, noms et descriptions de signaux utiles à connaître (on les trouve dans le fichier `/usr/include/signal.h`) :

2.2. PROCESSUS

1	HUP	(<i>hangup</i>) envoyé à ses processus quand on sort de la session de travail ou quand se déconnecte par un modem
2	INT	envoyé au processus en avant-plan dans le terminal dans lequel la touche d'interruption a été tapée (voir 3.3 et 3.7.1)
3	QUIT	comme INT, mais crée souvent un fichier core
9	KILL	signal qui ne peut être intercepté (voir commande <i>trap</i> en 17.5) et qui va donc tuer à coup sûr le processus qui le reçoit
15	TERM	(<i>terminate</i>) envoyé par défaut par la commande <i>kill</i> (voir 7.2). Pour terminer un processus d'une manière élégante si possible
18	TSTP	suspend un processus (touche Ctrl Z)
19	CONT	pour continuer après une suspension

2.2.2 Terminal de contrôle d'un processus, exécution en arrière-plan

Les processus lancés par un utilisateur restent liés au terminal d'où ils ont été lancés. On appelle ce terminal le terminal de contrôle du processus. Le système peut ainsi repérer les processus qui recevront le signal engendré par la frappe de la touche d'annulation ou les processus qui seront arrêtés si l'utilisateur éteint son terminal.

Un utilisateur peut lancer certains processus en arrière-plan (voir 14.5). Ces processus deviennent indépendants du terminal : ils s'exécutent sans intervention de l'utilisateur, sans que celui-ci ait à attendre la fin de leur exécution. Il peut ainsi lancer plusieurs tâches en parallèle (mais sans réelle synchronisation entre elles). Ces processus ne seront pas arrêtés si l'utilisateur appuie sur la touche d'annulation mais ils peuvent toujours être tués par la commande *kill* (voir 7.2). Ils ne peuvent pas lire les caractères tapés sur le terminal mais ils peuvent toujours y écrire.

Des processus peuvent être lancés automatiquement par le système, en arrière-plan (comme le système de spool pour l'impression étudié en 8.4). Ces processus sont appelés des *daemons*, acronyme de l'anglais "*deferred auxiliary execution monitor*" c'est-à-dire programme de contrôle d'exécution différée. On les appelle souvent "démons" en français.

2.2.3 Propriétaires et groupes effectifs et réels d'un processus

Chaque processus a un propriétaire effectif et un propriétaire réel. Le propriétaire réel est l'utilisateur qui a lancé la commande qui a généré les processus. Le propriétaire effectif détermine les droits du processus dans le système et en particulier les actions autorisées sur les fichiers (voir 2.4).

Le plus souvent le propriétaire effectif est le propriétaire réel. Il existe une exception importante liée à la notion de "*set user id*" étudiée en 2.4.2.

De même, chaque processus a un groupe réel (le groupe de l'utilisateur qui a lancé la commande qui a généré les processus) et un groupe effectif (voir 2.4.2). Nous n'étudierons pas ces notions en détails dans ce cours.

2.2.4 Création d'un nouveau processus, arbre des processus

Chaque processus (sauf le processus de pid 1) est créé par son processus parent. On a ainsi un arbre généalogique des processus créés dans une session. Les pid ainsi que les pid des parents de tous les processus sont affichés par la commande *ps*.

2.3 Fichiers

Les données (que l'on veut conserver entre deux sessions) et les programmes sont enregistrés dans des fichiers.

On a vu en 1.6 que Unix utilise aussi la notion de fichier pour organiser l'ensemble des fichiers (répertoires) et pour travailler avec les périphériques (fichiers spéciaux).

2.3.1 i-node

Outre le contenu du fichier, le système conserve d'autres informations (par exemple le propriétaire) sur chacun des fichiers de l'arborescence.

Ces informations lui seront utiles pour la gestion et la sécurité du système. Ces informations ne sont pas enregistrées avec le contenu du fichier, elles sont enregistrées dans la table des i-nodes (une table pour chaque système de fichiers) qui est conservée dans un endroit spécial sur le disque. Chaque fichier a un numéro de i-node (un nombre entier) qui indique dans quelle entrée de la table des i-nodes les informations système le concernant sont rangées.

2.3.2 Structure interne des répertoires

La liaison entre les noms des fichiers et les informations enregistrées dans le i-node du fichier se fait grâce aux répertoires.

Les répertoires sont des fichiers de l'arborescence mais leur contenu diffère des fichiers ordinaires. Un fichier répertoire contient le nom et le numéro de i-node des fichiers placés directement sous ce répertoire dans l'arborescence des fichiers.

Les informations concernant un fichier sont donc conservées dans trois endroits différents :

- le nom du fichier est conservé dans le répertoire contenant le fichier,
- les informations système sont conservées dans le i-node du fichier,
- le contenu du fichier (les données) est enregistré dans les blocs du disque dont les adresses sont dans le i-node.

2.3.3 Propriétaire et groupe d'un fichier

Au moment de sa création, le propriétaire d'un fichier est l'utilisateur qui l'a créé. Seul un super-utilisateur (voir 2.1.2) peut changer le propriétaire d'un fichier.

Un fichier a aussi un groupe qui est le nom d'un groupe d'utilisateur. C'est le plus souvent le groupe de l'utilisateur qui a créé le fichier. Cette notion de groupe facilite le partage de fichiers entre plusieurs utilisateurs. Nous n'étudierons pas cette notion en détail dans ce cours.

2.3.4 Mode d'accès au fichier

A chaque fichier sont attachées les *autorisations* pour le propriétaire du fichier, pour les membres du groupe du fichier (mais sans le propriétaire), pour les autres ("le reste du monde", ceux qui ne sont pas le propriétaire et qui n'appartiennent pas au groupe du fichier). Ces autorisations forment le *mode d'accès* au fichier. Elles sont affichées par la commande "**ls -l**" (voir figure 4.2 et section 4.2) juste après le type du fichier.

Les autorisations peuvent être (leur signification exacte sera étudiée en 2.4.1) :

- l'autorisation de lecture ("r" pour la commande "**ls -l**")
- l'autorisation d'écriture ("w" pour la commande "**ls -l**")
- l'autorisation d'exécution ("x" pour la commande "**ls -l**")

Il existe aussi des autorisations spéciales réservées à certains fichiers exécutables (elles accordent des droits spéciaux aux processus qui exécutent ces fichiers; elles sont étudiées en détail en 2.4.2) :

- *set user id* ; *suid* en abrégé ("s" à la place du "x" dans les autorisations du propriétaire pour la commande "**ls -l**")
- *set group id* ; *sgid* en abrégé ("s" à la place du "x" dans les autorisations du groupe pour la commande "**ls -l**")

2.4 Protection des fichiers

La protection des fichiers en Unix est fondée sur les droits des processus sur les fichiers (rappelons que le processus est l'unité élémentaire d'exécution en Unix). Un processus n'est pas autorisé à faire n'importe quelle action sur les fichiers. Par exemple, il peut avoir le droit de lire le contenu d'un fichier mais pas de supprimer ce fichier. Nous allons examiner dans cette section comment sont déterminés les droits d'un processus sur un fichier.

2.4.1 Droits associés aux autorisations

Les autorisations qui constituent le mode d'accès au fichier ne définissent pas toujours d'une façon évidente les actions qu'un processus a le droit d'exécuter sur le fichier. Par exemple, un processus peut avoir le droit de supprimer un fichier sans avoir l'autorisation "w" sur ce fichier. On peut aussi s'interroger sur la signification de l'autorisation "x" pour un répertoire.

L'autorisation de lecture correspond au droit de lire le contenu du fichier, l'autorisation d'écriture correspond au droit d'écrire dans le fichier, l'autorisation d'exécution correspond au droit d'exécution du fichier si le fichier n'est pas un répertoire et au droit de travailler dans son arborescence si le fichier est un répertoire.

Pratiquement, on a le tableau suivant :

Type de Fichier	Autorisation	Fichier ordinaire	Répertoire
Lecture	r	lire le contenu du fichier	listier les sous-fichiers
Ecriture	w	modifier le contenu du fichier	ajouter, enlever des sous-fichiers
Exécution	x	exécuter le fichier	travailler dans le répertoire

Pour mieux comprendre le tableau ci-dessus il est bon de se rappeler la structure interne d'un répertoire (voir 2.3.2). Ainsi, on comprend que, si on veut supprimer un fichier d'un répertoire, il faut et il suffit d'avoir l'autorisation d'écrire dans le *répertoire* (pour effacer l'entrée correspondant au fichier dans le répertoire).

Donc, si l'on veut protéger un fichier sans interdire l'accès au répertoire dans lequel il est situé, il faut enlever l'autorisation d'écriture dans le répertoire parent pour interdire la suppression de ce fichier et enlever l'autorisation d'écriture sur le fichier lui-même pour interdire la modification du fichier.

2.4.2 Autorisations "set user id" et "set group id"

Les droits d'un processus sur un fichier dépendent du propriétaire et du groupe effectifs d'un processus et pas du propriétaire et du groupe réels.

Habituellement le propriétaire et le groupe effectifs sont le propriétaire et le groupe réels (l'utilisateur qui a lancé le processus, et son groupe). Cependant, si le code exécuté par un processus est celui d'un fichier binaire qui a le bit "*set user id*" (resp. "*group user id*") positionné, le propriétaire effectif (resp. le groupe effectif) du processus est le propriétaire (resp. le groupe) du fichier binaire.

Ce fait est par exemple utilisé par la commande *passwd* pour permettre à un simple utilisateur de changer son mot de passe dans des fichiers qu'il n'est pas normalement autorisé à modifier : seul le super-utilisateur "root" a le droit de modifier le fichier des mots de passe mais la commande *passwd* a l'autorisation *set user id* et appartient à root. Le simple utilisateur a donc tous les droits de *root* le temps qu'il exécute la commande *passwd* (qui ne lui permet de modifier que son propre mot de passe). Il récupère ses droits de simple utilisateur dès la fin de l'exécution de la commande *passwd*.

2.4.3 Mécanisme de protection des fichiers

On a vu en 2.4.1 les actions qu'un processus a le droit d'effectuer sur un fichier ; elles dépendent des autorisations que ce processus a sur le fichier lui-même et sur les répertoires parents de ce fichier. Voici les règles qui déterminent les autorisations d'un processus sur un fichier :

- si le propriétaire effectif du processus et le propriétaire du fichier sont les mêmes, les autorisations du processus sont les autorisations du fichier accordées au propriétaire
- sinon, si le groupe du fichier est le groupe effectif du processus ou l'un des groupes de l'utilisateur fichier, les autorisations du processus sont les autorisations du fichier accordées au groupe,
- sinon, les autorisations du processus seront celles accordées aux "autres".

Chapitre 3

Premiers pas dans le système

Ce chapitre donne quelques informations utiles pour commencer à travailler dans un système Unix. Certaines notions évoquées sont développées dans la suite du cours (par exemple, les fichiers de configuration du shell).

3.1 Entrée dans le système

Lorsqu'un terminal est allumé, certains processus sont automatiquement lancés par le système pour gérer les entrées et les sorties depuis ce terminal. En particulier, un processus est lancé pour attendre l'entrée d'un nom d'utilisateur et d'un mot de passe.

Le nom d'utilisateur est attribué par l'administrateur du système.

3.1.1 Changer son mot de passe (passwd)

Par prudence, l'utilisateur doit changer le mot de passe que l'administrateur lui a attribué lorsqu'il entre pour la première fois dans le système.

`passwd` permet d'entrer un nouveau mot de passe.

Remarque 3.1

Sur les systèmes dont les mots de passe sont gérés par le système NIS (voir 6.6), les mots de passe (et plusieurs autres fichiers utilisés par le système) de plusieurs machines sont centralisés sur une seule machine. La commande pour changer le mot de passe est alors *yppasswd* qui a la syntaxe :

```
yppasswd nom-login
```

L'utilisateur doit entrer l'ancien mot de passe avant de donner le nouveau. Pour éviter des fautes de frappe (impossibles à voir puisque le mot de passe ne s'affiche pas à l'écran), le nouveau mot de passe doit être entré deux fois. Seul le propriétaire du mot de passe (ou le super utilisateur) peut entrer un nouveau mot de passe. Le mot de passe doit comporter au moins 6 caractères.

3.2. SORTIE DU SYSTÈME

Dans un mot de passe on peut utiliser tous les caractères sauf # et @. Dans les commandes pour tester si un mot de passe est valable, les majuscules et les minuscules ne sont pas équivalentes.

Pour des raisons de sécurité, l'administrateur peut imposer la modification du mot de passe à intervalles réguliers.

3.1.2 Démarrage d'une session

Après que l'utilisateur ait entré son nom et son mot de passe, certaines actions sont automatiquement effectuées par Unix :

- le shell de démarrage est lancé (le plus souvent *zsh* pour les étudiants de l'université de Nice),
- le répertoire HOME devient le répertoire courant,
- des fichiers d'initialisation sont automatiquement lancés. Ils initialisent l'environnement de l'utilisateur. Ils donnent en particulier des valeurs initiales aux variables TERM (nom du type de terminal, le plus souvent "xterm" si on travaille sous X Window) et PATH (voir 3.7.2). Ces fichiers sont étudiés plus en détails en 15.3.

Remarque 3.2

Si on travaille sous X Window, une variable importante est DISPLAY qui indique aux clients X (programmes qui utilisent le système graphique X Window) avec quel terminal clavier-écran-souris ils doivent travailler.

3.2 Sortie du système

La commande *exit* permet de sortir d'un shell. S'il s'agit du shell de connexion, la session de travail se termine.

3.3 Touches spéciales

Quelques touches/caractères ont des significations spéciales (Ctrl C s'obtient au clavier en maintenant appuyée la touche [Ctrl] du clavier tout en appuyant sur la touche C) :

- Ctrl S** interrompt la transmission des caractères entre le terminal et le programme en cours
- Ctrl Q** rétablit la transmission en "libérant" tous les caractères tapés depuis un précédent Ctrl S
- Ctrl D** est transmis comme une fin de fichier par le driver de terminal s'il est l'unique caractère d'une ligne (voir 8.1). Elle permet donc de terminer un shell à la place de la commande *exit*, sauf si l'utilisateur inhibe cette fonction pour éviter des sorties involontaires en plaçant

la commande interne “`set -o ignoreeof`” du shell dans un fichier d’initialisation du shell (voir 15.3). Il est en effet facile de taper par erreur [Ctrl] D au lieu, par exemple, de [Shift] D.

La touche [Ctrl] D permet aussi d’effacer le caractère sur lequel se trouve le curseur quand on modifie une ligne de commande.

Ctrl C interrompt l’exécution d’une commande

Ctrl U efface la ligne de commande en cours d’écriture

Les quatre flèches permettent de rappeler les commandes déjà tapées et de se déplacer dans ces commandes pour les modifier (tout caractère tapé s’insère dans la commande).

3.4 Format des commandes Unix

Une grande partie des commandes Unix ont été écrites à l’origine par des utilisateurs et ont été intégrées dans le lot des commandes standards. Dans un premier temps, aucune syntaxe standard n’avait été clairement choisie pour le format des options des commandes. Une conséquence fâcheuse est que certaines options qui se correspondent dans des commandes différentes, peuvent avoir des noms et des syntaxes différentes. Par exemple, pour désigner un séparateur de champ, la commande “*sort*” utilise “-d” alors que la commande “*cut*” utilise “-t”.

La syntaxe générale des commandes Unix est :

```
commande options... arguments...
```

Les *arguments* indiquent les objets sur lesquels la commande va agir et les *options* indiquent des variantes dans l’exécution de la commande.

Les options sont le plus souvent précédées d’un tiret “-”. L’ordre des options est le plus souvent indifférent et plusieurs options peuvent être regroupées derrière un seul tiret.

Les arguments peuvent être absents et, dans ce cas, prennent des valeurs par défaut.

Exemples 3.1

- (a) `date`
- (b) `ls -la`
- (c) `ls -l /users/students`
- (d) `ls /users/students`

Dans les manuels de référence, “[]” indique les options ou les arguments facultatifs, “*arg...*” indique que l’argument *arg* peut être répété.

Exemple 3.2

```
ls [-abcCdFgilmnopqrRstux] [files...]
```

3.5 Consultation du manuel en ligne (man)

La commande *man* permet de consulter le manuel de référence des commandes Unix. `man -s [numéro-section] commande` affiche les pages du manuel de référence sur la commande et dans la section indiquée. Sur certaines versions de *man* le numéro de la section est donné seul, sans être précédé de l’option “-s”. “1” est le numéro de section par défaut.

Les différentes sections sont les suivantes (sur un Unix Solaris; on peut trouver des variantes sur les différents Unix) :

1. commandes pour l’utilisateur (ls, sort, etc.),
2. appels système (pour les programmeurs en langage C),
3. fonctions diverses de différentes bibliothèques (pour les programmeurs en langage C),
4. information sur les formats de fichiers (passwd, group, etc.),
5. informations diverses,
6. jeux et démonstrations,

Les informations pour l’administration du système sont données dans la section 1M.

Chaque section a une entrée particulière “intro” qui la décrit et donne quelques informations générales.

Exemples 3.3

- (a) `man man`
- (b) `man ls`
- (c) `man -s 2 intro` (ou `man 2 intro` suivant les Unix)

```
man -k mot-clé
```

affiche une ligne sur chaque entrée (en général le nom d’une commande Unix) du manuel qui contient *mot-clé* dans la section “NAME” (qui comporte une description succincte de la commande) du manuel.

Remarque 3.3

Pour que cette option fonctionne, l’administrateur doit effectuer une indexation préalable des noms de commandes qui sont répertoriées dans le manuel.

Exemple 3.4

```
man -k directory
```

3.6 Nom d’un fichier, nom absolu, nom relatif

Le *nom terminal* est composé d’au plus 14 caractères si l’on veut rester compatible avec toutes les versions d’Unix système V (jusqu’à 255 caractères dans toutes les versions récentes d’Unix). Tous les caractères du code ASCII sont autorisés sauf “/” et le caractère de code 0.

Le *nom (ou chemin) absolu* est composé du nom terminal précédé du chemin d’accès à ce fichier depuis la racine. Par exemple, “/users/students/jean/affiche” Le premier “/” désigne le répertoire racine. Les “/” suivants servent de séparateurs entre les noms des répertoires et le nom terminal.

Ce nom absolu est souvent lourd à manier car il peut être très long. Un fichier peut être désigné par son *nom (ou chemin) relatif* qui correspond à la position du fichier relativement au répertoire courant où se trouve le processus qui désigne le fichier (voir 2.2). Un nom est relatif quand il ne commence pas par un “/”.

Exemple 3.5

Si l’utilisateur (plus précisément le processus qui exécute le shell courant de l’utilisateur) se trouve dans le répertoire “/users/students/jean”, il pourra désigner le fichier de l’exemple ci-dessus par “affiche”. S’il se trouve dans le répertoire “/users/students”, il pourra désigner ce fichier par “jean/affiche”.

Noms relatifs particuliers :

- .. désigne le répertoire courant (où l’on se trouve)
- .. désigne le répertoire parent du répertoire courant

3.7 Commandes

3.7.1 Lancer une commande, supprimer un processus

Lorsqu’il a la main (le prompt est affiché) l’utilisateur peut lancer une commande en tapant son nom suivi éventuellement de ses options et de ses arguments.

La plupart des commandes peuvent s’interrompre en tapant sur la touche d’annulation. Cette touche est le plus souvent [Ctrl] C ou la touche [DEL].

Certaines commandes peuvent se protéger contre une interruption par la touche d’annulation. Dans ce cas, on peut tout de même les interrompre par la commande *kill* (voir 7.2).

Un processus lancé en arrière-plan est automatiquement protégé d’une interruption par la touche d’annulation et la seule façon de l’interrompre est d’utiliser la commande *kill*.

3.7.2 Nom d’une commande, variable PATH

Pour faciliter le travail de l’utilisateur, les commandes qui sont situées dans la liste des répertoires enregistrée dans la variable PATH (attention, le nom est en majuscules)

3.7. COMMANDES

peuvent être appelées par leur nom terminal. Il n’est pas nécessaire de taper leur nom absolu ou relatif.

On peut par exemple taper ‘man’ au lieu de ‘/bin/man’ parce que le répertoire /bin fait partie de la liste de répertoires de la variable PATH.

On peut afficher la valeur de PATH avec la commande (en général la liste est plus longue que la réponse donnée ici ; “(”) est le prompt) :

```
(~) echo $PATH
/bin:/usr/bin:/users/students/toto/bin:/usr/ucb:.
```

La valeur est une suite de noms de répertoires séparés par “:”. Le répertoire courant peut être indiqué par un “.”, ou par “:” au début ou à la fin de la liste.

Pour des raisons de sécurité, il n’est pas conseillé de mettre le répertoire courant au début de la liste. En effet, la liste est parcourue de gauche à droite pour rechercher une commande et on n’est pas certain de ne pas avoir par exemple un fichier “ls” piégé dans le répertoire courant, lorsque l’on se déplace dans l’arborescence et que l’on lance cette commande.

Cette variable est initialisée par l’administrateur du système au moment de la création de l’utilisateur. L’utilisateur peut ensuite modifier sa valeur. Le plus souvent cette valeur est donnée dans le fichier .zshenv si on travaille avec zsh (voir 15.3).

Remarque 3.4

La variable PATH n’est utilisée pour rechercher une commande que si le nom de la commande ne comporte aucun “/”. Si vous tapez “bin/cmd”, la commande ne sera pas recherchée que dans le répertoire bin situé sous le répertoire courant et pas sous tous les répertoires bin de tous les répertoires listés dans la variable PATH. Taper “./cmd” est un bon moyen d’exécuter une commande de son répertoire courant, alors que “cmd” risque d’exécuter un autre “cmd” situé dans un des répertoires de la variable PATH.

3.7.3 Nom complet et type d’une commande (whence, whereis)

Comme on vient de le voir, la facilité offerte par la variable PATH provoque parfois des problèmes. Sous *ksh* et *zsh* (mais pas sous *sh*), la commande *whence* permet de savoir où se situera la commande exécutée :

```
whence [-vp] commande
```

indique comment serait interprétée la commande si elle était tapée : fonction, commande interne du shell, alias ou chemin absolu pour un fichier situé dans un des répertoires de la variable PATH.

-v est l’option “verbeuse” un peu plus “bavarde”. Avec l’option -p, on demande de ne rechercher que les fichiers exécutables (et pas les fonctions, commandes internes ou alias).

Certaines commandes peuvent se retrouver à plusieurs endroits de l’arborescence. Ce sont souvent des versions différentes. On peut par exemple avoir les versions Unix BSD et Unix OSF d’une commande. Pour retrouver toutes les versions qui se trouvent à des emplacements standards de l’arborescence, on dispose de la commande *whereis*. Cette

commande donne aussi les emplacements de pages du manuel en ligne et des sources des programmes.

3.7.4 Complétion des commandes par `zsh`

Avec `zsh`, l'utilisateur peut utiliser la touche [Tab] pour compléter les noms de commandes ou les noms de fichiers.

S'il y a plusieurs possibilités, `zsh` complète avec le plus de caractères qu'il peut et envoie un “beep” sonore. L'utilisateur peut avoir les différentes possibilités en tapant Ctrl D.

S'il ne peut compléter, il envoie un “beep” sonore.

Pour entrer une tabulation dans une commande, l'utilisateur doit la faire précéder de Ctrl V.

Deuxième partie

Commandes

Chapitre 4

Commandes liées à l’arborescence des fichiers

4.1 Visualisation de l’arborescence (ls)

```
ls [options...] [fichiers...]
```

donne des informations pour chaque fichier spécifié (pour le répertoire en cours si aucun fichier n’est spécifié). Le type d’information donné dépend du type du fichier :

- si le fichier cité est un répertoire, ls affiche des renseignements sur les fichiers (de tous types, pas seulement les fichiers ordinaires) qui sont directement sous ce répertoire. Sans option, seuls les noms terminaux (pas les noms absolus ou relatifs) des fichiers sont donnés.
- si un fichier cité n’est pas un répertoire, ls affiche des renseignements sur ce fichier. Le nom du fichier est donné (tel qu’il a été donné dans la ligne de commande : nom relatif ou absolu). Sans option, c’est surtout utile quand on emploie des caractères spéciaux (* ou ? par exemple, étudiés en 4.5) dans la ligne de commande.

Options

- l** format détaillé (voir 4.2)
- a** liste aussi les noms de fichiers qui commencent par un “.” (qui ne sont pas normalement listés)
- A** comme l’option -a mais les répertoires “.” et “..” ne sont pas affichés
- d** si l’argument est un répertoire, liste seulement son nom (et pas les fichiers qui sont sous ce répertoire)
- t** trie par date de dernière modification (sinon la liste des fichiers est triée par défaut par ordre alphabétique), les plus récemment modifiés en premier
- i** affiche au début de la ligne le numéro de i-node des fichiers
- R** listage récursif des sous-répertoires
- g** affiche le groupe du fichier quand elle est associée à l’option “-l” (utile seulement en Unix BSD ; voir remarque (a) de 4.2.1)

```
$ ls -l /
total 20954
lrwxrwxrwx  1 root    system      7 Apr 26  1995 bin -> usr/bin
drwxr-xr-x  6 root    system    6144 Jan 18  07:02 dev
drwxr-xr-x 15 root    system   4608 Jan 18  07:02 etc
lrwxrwxrwx  1 root    system      7 Apr 26  1995 lib -> usr/lib
drwxr-xr-x  2 root    system     512 Feb  3  1995 mnt
drwxr-xr-x  4 root    system     512 Sep 19 10:52 net
drwxr-xr-x 10 root    system    2560 Jun 23  1995/sbin
drwxr-xr-x 25 root    system     512 Dec  7 10:15 students
drwxrwxrwt  3 root    system    5120 Jan 18  08:45 tmp
drwxr-xr-x  4 root    system     512 Jan 13  1995 users
drwxr-xr-x 22 root    system     512 Dec 19 13:07 usr
drwxr-xr-x 20 root    system     512 Apr 26  1995 var
-rwxr-xr-x  1 root    bin      9246472 Jun 23  1995 vmunix
.....
```

FIG. 4.1 – Format de sortie de la commande “ls -lg”

-L si un fichier est un lien symbolique, cette option indique que l’on veut des renseignements sur le fichier pointé par le lien et pas sur le fichier lui-même (voir commande *ln* en 9.3)

Exemples 4.1

- (a) `ls -l bin /usr/bin`
- (b) `ls -ld a*`
affiche les informations sur les fichiers du répertoire courant (dont le nom commence pas par un “a”). L’option “-d” est indispensable pour ne pas “entrer” dans les sous-répertoires s’il y en a dont le nom commence par “a”.
- (c) `ls -t f* | head -1`
affiche le nom du fichier le plus récemment modifié parmi tous les fichiers du répertoire courant, dont le nom commence par un “f”.

4.2 Information détaillée sur les fichiers (ls -l)

4.2.1 Format d’affichage

Un exemple de format d’affichage de la commande `ls -l` appliquée à un répertoire est donné à la figure 4.1. Une première ligne donne le nombre de blocs de 1024 octets qu’occupent l’ensemble des fichiers listés par la commande. Le format d’une ligne est détaillé dans la figure 4.2.

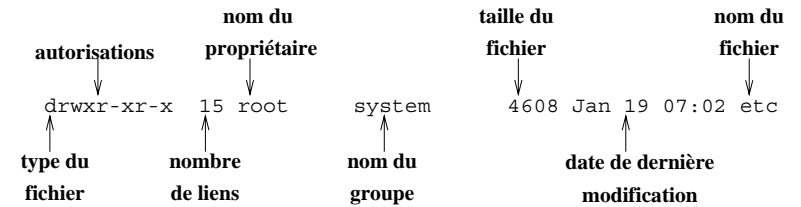


FIG. 4.2 – Format de sortie d’une ligne de la commande “ls -lg”

Remarques 4.1

- (a) Les versions Unix OSF et Système V de *ls* affichent le nom du groupe du fichier avec l’option “-l”. En Unix BSD, le groupe s’affiche avec l’option “-g”. Si on veut récupérer une information dans une des colonnes il est donc préférable d’associer toujours l’option “-g” à l’option “-l”.
- (b) Les dates sont affichées avec l’année, ou l’heure et la minute selon que la date est plus ou moins récente (six mois avant la date actuelle).

4.2.2 Types de fichiers

Le type du fichier est affiché à l’aide du premier caractère de la ligne de description du fichier :

- fichier ordinaire
- d** répertoire (directory)
- b** fichier spécial de type bloc (essentiellement les disques)
- c** fichier spécial de type caractère (disque, streamer, terminal, etc.)
- l** lien symbolique
- s** socket

Pour les liens symboliques, le nom du fichier pointé est donné à la suite des autres renseignements sur le fichier :

```
lrwxrwxrwx  1 root    1200  Feb 25 20:09 fich -> fichierP
```

Pour les fichiers spéciaux, la taille n’a aucun sens ; elle est remplacée par le majeur et le mineur : le majeur désigne le pilote (le programme de gestion du périphérique ; *driver* en anglais) et le mineur désigne le périphérique particulier géré par ce pilote :

```
crw-rw-rw-  1 root    39,6  Feb 25 20:09 tty01
```

4.2.3 Mode d’accès au fichier

Les autorisations (on dit aussi le mode d’accès au fichier) du fichier sont indiquées par les neuf caractères qui suivent le type du fichier. Les trois premiers caractères sont les autorisations du propriétaire du fichier, les trois suivants sont les autorisations du groupe d

fichier et les trois derniers sont les autorisations des autres (qui ne sont pas le propriétaire et qui n'appartiennent pas au groupe).

4.2.4 Nombre de liens

Pour les fichiers ordinaires il s'agit du nombre de fichiers ayant le même numéro de i-node.

Pour les répertoires, il indique le nombre de sous-répertoires du répertoire (augmenté de 2, car . et .. sont comptés).

4.3 Se déplacer dans l'arborescence (cd)

```
cd [répertoire]
```

déplace le processus dans *répertoire* (répertoire “HOME” par défaut).

Exemple 4.2

```
cd /users/students/jean
```

Une facilité offerte par *zsh* est bien utile quand on a besoin de travailler entre deux répertoires :

```
cd -
```

permet de se déplacer dans le répertoire où l'on était avant le dernier déplacement.

4.4 Afficher le répertoire courant (pwd)

```
pwd
```

Il est souvent plus prudent de taper cette commande avant d'effectuer une commande dangereuse (suppression de fichiers par exemple) pour savoir où l'on est placé dans l'arborescence.

4.5 Caractères spéciaux pour le shell

4.5.1 Génération des noms de fichiers (*, ?, [])

Les caractères “* ? []” sont interprétés par le shell quand ils figurent dans une commande. Cette interprétation s'appelle la génération des noms de fichiers. Le shell interprète chaque mot qui les contient comme un modèle de noms de fichiers : il remplace le mot par une liste alphabétique des noms des fichiers qui correspondent à ce modèle.

Si aucun fichier ne correspond au modèle, le modèle est laissé tel quel par *sh* et *ksh*. Pour *zsh*, c'est le cas si l'option `NO_NOMATCH` est positionnée (voir 14.1.2), sinon, *zsh* interrompt l'interprétation de la commande et ne lance pas son exécution.

* désigne zéro, un ou plusieurs caractères,

4.5. CARACTÈRES SPÉCIAUX POUR LE SHELL

? désigne un caractère quelconque,

[c₁-c₂] ou [c₁c₂...] désigne un caractère spécifié par l'intérieur des crochets. Par exemple, [a-z]

désigne une lettre minuscule et [AEIOUY] désigne une voyelle majuscule.

On peut aussi désigner un caractère quelconque qui n'est pas parmi les caractères spécifiés entre les crochet en faisant suivre le crochet ouvrant du caractère “~”

Par exemple, sous *zsh*, “[~0-9]” désigne un caractère qui n'est pas un chiffre.

Attention, sous *sh* et *ksh*, il faut écrire “[!0-9]”¹.

\ supprime la signification spéciale du caractère suivant.

Exemple 4.3

Si l'utilisateur tape la commande

```
ls -l bin/A[0-9]*
```

la commande suivante sera lancée après l'interprétation du shell :

```
ls -l bin/A1abc bin/A2xy
```

si les 2 fichiers A1abc et A2xy sont les deux seuls fichiers du répertoire bin qui commencent par un A suivi d'un chiffre.

Remarque 4.2

Attention, “*” ne peut désigner une chaîne de caractères qui comprend un “/” ou une chaîne de caractères qui commence par un “.” si ce “.” est placé en tête ou juste derrière un “/”. Ces deux exceptions sont aussi valables pour les caractères spéciaux “?” et “[]”.

Autrement dit, le caractère “.” doit être explicitement désigné s'il est placé au début d'un nom terminal de fichier. Par exemple, si on veut tous les fichiers du répertoire courant qui se terminent par la lettre c (y compris ceux qui commencent par un “.”) il faut taper

```
ls -d .*c *c
```

L'interprétation par le shell de ces caractères spéciaux peut avoir des effets indésirables car le shell l'effectue n'importe où dans une commande. On peut utiliser \ pour éviter cette interprétation sur le caractère suivant. On peut aussi utiliser " ou ' pour empêcher l'interprétation sur une portion de texte (voir 14.9).

4.5.2 Le caractère spécial ~

Pour *ksh* et *zsh* (mais pas pour *sh*), le caractère “~” (seul ou placé en tête de mot) a une signification particulière :

~ désigne le répertoire HOME de l'utilisateur,

~*utilisateur*

désigne le répertoire HOME de l'*utilisateur* indiqué.

¹ Il faut se souvenir (voir 14.11.1) que, sous *zsh* et sans indication particulière, un shellsript est exécuté par *sh* et il faut donc utiliser cette syntaxe dans les shellscripts

Exemples 4.4

- (a) `cp ~/fich ~/bin`
- (b) `ls -l ~/toto/bin/fich`

4.5.3 Le caractère spécial #

Dans les shells, le shell considère que tout mot qui commence par un “#” est le début d’un commentaire qui va jusqu’à la fin de la ligne. Si un mot commence par un “#” et n’introduit pas un commentaire, il faut donc le faire précéder de “\”. Un “#” à l’intérieur d’un mot n’est pas considéré comme un caractère spécial.

Remarques 4.3

- (a) En mode interactif (c’est-à-dire lorsque la commande est tapée directement au clavier par l’utilisateur), *zsh* (mais pas *ksh*) ne considère pas “#” comme un caractère spécial.
- (b) Voir le cas particulier d’un fichier exécutable dont la première ligne commence par “#!” en 14.11.1.

4.5.4 Autres caractères spéciaux

Sous *zsh*, ! et la tabulation sont des caractères particuliers.

! est lié à l’historique des commandes. Si on veut l’utiliser dans un autre sens, il faut le faire précéder de \. Ce caractère étant lié à l’historique des commandes n’a d’utilité qu’en interactif. *zsh* ne le considère donc pas comme un caractère spécial dans un shellscript.

La tabulation permet la complétion de commande. Si on veut insérer une tabulation dans une commande, on la fait précéder de “Ctrl V”.

4.6 Afficher le type d’un fichier (file)

```
file fichiers...
```

tente de deviner le type de *fichier* (répertoire, exécutable binaire, shellscript, ASCII, langage, etc.).

4.7 Rechercher des fichiers dans l’arborescence (find, locate)

```
find répertoires... -name modèle-nom -print
```

affiche les noms des fichiers dont les noms correspondent à *modèle-nom* et situés dans l’arborescence des *répertoires*.

modèle-nom peut comporter des caractères spéciaux semblables aux caractères spéciaux utilisés dans la génération des noms de fichiers par le shell (voir 4.5). Plus exactement, find

4.7. RECHERCHER DES FICHIERS DANS L'ARBORESCENCE (FIND, LOCATE)

reconnait les caractères *, ?, [] (attention, la négation des caractères entre crochets est donnée par un ! et pas par un ^). Un “.” n’est reconnu en début de nom que s’il est donné explicitement.

Exemple 4.5

```
find /usr -name "info*" -print
```

recherche tous les fichiers situés sous /usr dont le nom commence par info. On doit entourer info* avec des guillemets pour éviter le remplacement (par le shell) de * par des noms de fichiers (voir 14.8).

La commande find a de nombreuses autres options qui permettent de rechercher les fichiers qui vérifient certains critères. En voici quelques unes :

- size *n*** taille en nombre de blocs de 512 octets
- mtime *n*** nombre de jours depuis la dernière modification
- atime *n*** nombre de jours depuis le dernier accès
- Dans les options ci-dessus, *n* peut être remplacé par +*n* (nombre supérieur à *n*) ou par -*n* (nombre inférieur à *n*).
- user *nom*** propriétaire
- type *t*** type du fichier (un “f” pour un fichier ordinaire et les autres abréviations) comme dans la commande *ls -l*; voir 4.2)

- perm *nb _octal***
est vrai si le fichier a les autorisations *nb _octal*
- perm -*nb _octal***
est vrai si le fichier a *au moins* les autorisations *nb _octal*

Les deux options “-exec” et “-ok” permettent d’exécuter une commande sur les fichiers qui vérifient les critères précédents. Le fichier examiné peut être désigné par “{” (isolé des autres caractères) dans la commande. Ces deux options renvoient “vrai” si le code retour de la commande est 0 (voir 14.1.6). L’option “-ok” demande une confirmation de l’utilisateur avant l’exécution de la commande. Il ne faut pas oublier le “;” final qui indique que la commande est finie. Il doit être précédé par “\” pour que le shell ne l’interprète pas.

- exec *commande* \;**
- ok *commande* \;**

Toutes ces options peuvent être reliées par les opérateurs logiques “!” (négation), “-” (ou logique) et regroupées avec des parenthèses. Le *et* logique est obtenu par la juxtaposition des options.

Les options sont examinées (et exécutées pour -print, -exec et -ok) tant que la condition peut être satisfaite pour le fichier en cours d’examen. Dès que la condition ne peut plus être satisfaite find passe au fichier suivant.

Exemples 4.6

- (a) `find ~ -mtime -10 -print`
affiche les noms des fichiers de l’arborescence du répertoire HOME de l’utilisateur qui ont été modifiés dans les dix derniers jours.

- (b) `find /users -user toto -mtime -10 -exec ls -ld {} \;`
affiche des renseignements détaillés sur les fichiers de l'arborescence de /users, qui appartiennent à toto et qui ont été modifiés dans les 10 derniers jours.
- (c) `find ~toto ! -user toto -exec ls -ld {} \;`
affiche des renseignements détaillés sur les fichiers qui n'appartiennent pas à toto dans son répertoire HOME.
- (d) `find ~toto -perm -002 -exec ls -ld {} \;`
affiche des renseignements détaillés sur les fichiers du répertoire HOME de toto, qui ont (au moins) l'autorisation d'écriture pour les autres.
- (e) `find ~ -name core -ok rm {} \;`
supprime (après confirmation par l'utilisateur) tous les fichiers "core" situés sous le répertoire HOME.

Les fichiers "core" sont créés automatiquement quand certains programmes s'interrompent par suite d'une erreur. Ils peuvent servir aux experts pour trouver la cause de l'erreur.
- (f) `find . \(-name "*.java" -o -name "*.html" \) -exec more {} \;`
affiche page à page tous les fichiers "*.java" et "*.html" de l'arborescence du répertoire courant.

`locate modèle-de-nom...`

recherche les fichiers dont le nom correspond à *modèle-de-nom*.

Si *modèle-de-nom* contient des "jokers" (*, ? ou |), *locate* donne tous les fichiers dont le nom contient *modèle-de-nom*. Sinon elle donne les fichiers dont le nom correspond au modèle.

Cette commande GNU n'est pas toujours installée sur les systèmes Unix. Si l'administrateur l'a installée, elle permet de gérer une base de données des fichiers disponibles. Dans ce cas, cette commande est énormément plus rapide que la commande *find* et charge beaucoup moins le système.

Chapitre 5

Protection des fichiers

5.1 Changement des autorisations (chmod)

`chmod [-R] mode-accès fichiers...`

donne des autorisations aux fichiers. Seuls le propriétaire du fichier et le super utilisateur peuvent utiliser chmod.

mode-accès indique quelles sont les autorisations que l'on donne; on peut donner ces autorisations sous forme absolue ou symbolique.

L'option -R (récursif) indique que, si un des fichiers est un répertoire, chmod doit changer les autorisations de toute l'arborescence du répertoire.

Mode d'accès absolu

Les autorisations sont données par *mode-accès* sous forme d'un nombre octal composé de 3 chiffres.

Le premier chiffre correspond aux autorisations que l'on donne au propriétaire des fichiers, le deuxième correspond au groupe des fichiers et le troisième correspond à tous les autres utilisateurs.

Pour calculer chacun des chiffres, on ajoute les valeurs des autorisations en comptant 4 pour read, 2 pour write et 1 pour execute.

d'où, par exemple,

`chmod 755 fichier`

donne toutes les autorisations au propriétaire et les autorisations de lecture et d'exécution aux autres.

Mode d'accès symbolique

Les autorisations sont données par *mode-accès* sous la forme suivante :

[qui]... op permissions... [op permissions ...]

"qui" désigne celui ou ceux qui recevront ces autorisations : u (propriétaire), g (groupe) ou o (les autres) ou a (tous les 3, par défaut)

"op" indique si l'on veut ajouter ou enlever des autorisations : + (ajouter), - (enlever) = (donner les autorisations comme dans le mode absolu)

Les permissions sont données par r, w ou x.

Exemples 5.1

- (a) `chmod ug+wx fichier`
- (b) `chmod ug+w-x fich`
- (c) `chmod o=w fich`

5.2 Masque pour les autorisations (umask)

`umask [nnn]`

donne la valeur octale *nnn* (cf. mode absolu de `chmod` en 5.1) au masque qui sera utilisé pour limiter les autorisations que recevront les nouveaux fichiers au moment de leur création.

Exemple 5.2

`umask 026`

les fichiers et répertoires qui seront créés dans la suite de la session n'auront ni l'autorisation d'écriture pour le groupe (2) ni les autorisations d'écriture et de lecture pour les autres (6).

`umask`

(seul) affiche le masque en cours. Celui-ci est en général 022 si l'utilisateur n'a pas lancé de commande `umask`. Les nouveaux fichiers sont alors créés avec les autorisations "`rw-r--r--`" (car ils ne sont pas supposés être exécutables) et les nouveaux répertoires avec les autorisations "`rwxr-xr-x`".

Chapitre 6

Commandes d'observation du système

6.1 Date et Heure (date)

`date`

affiche la date et l'heure. Des formats d'affichage sont disponibles en option.

6.2 Nom de l'ordinateur (hostname)

`hostname`

affiche le nom de l'ordinateur sur lequel la commande s'exécute ("`taloo.unice.fr`", par exemple).

6.3 Nom du système d'exploitation (uname)

`uname -a`

affiche des informations sur le système d'exploitation.

6.4 Information sur les utilisateurs (who, finger)

`who`

affiche le nom de login, le nom du terminal et l'heure du login pour chaque utilisateur actuellement en ligne.

Si un utilisateur s'est connecté par l'intermédiaire du réseau (et pas directement sur la machine qui exécute la commande `who`), le nom de la machine d'où il vient est indiqué entre parenthèses à la fin de la ligne.

`who am I`

affiche les informations sur l'utilisateur qui s'est connecté pour démarrer la session en cours.

`whoami`

affiche le nom de login de l'utilisateur qui a lancé la commande.

```
finger [@machines...]
```

affiche les informations de la commande *who* et des informations informelles sur les utilisateurs (leur nom et prénoms, leur téléphone,...) actuellement connectés sur les *machines* (la machine locale par défaut).

```
finger utilisateur[@machine]...
```

affiche les informations sur les utilisateurs spécifiés, même s'il ne sont pas connectés. *finger* donne plus d'informations sur les utilisateurs que lorsque l'on n'indique pas d'utilisateurs en particulier. De plus les contenus des fichiers *.plan* et *.project* des répertoires HOME des *utilisateurs* sont affichés. Pour "*utilisateur*" on peut donner le nom de login ou le nom "réel" affiché par la commande.

```
rusers
```

devrait donner une liste des utilisateurs connectés sur le réseau (mais ça ne marche pas toujours...).

6.5 Dernières connexions au système (last)

Il est prudent de vérifier régulièrement que personne n'est entré sous son propre nom.

```
last [-n] [utilisateur] [terminal]
```

affiche des informations sur les dernières connexions. On peut se limiter aux connexions d'un *utilisateur* ou aux connexions établies à partir d'un certain *terminal*. On peut aussi se limiter aux *n* dernières connexions.

Exemple 6.1

```
last -3 toto
```

affiche les trois dernières connexions (y compris l'actuelle) de toto.

Remarque 6.1

last ne donne pas toujours les connexions à partir des terminaux X ou des stations de travail. Cela dépend du système de gestion des connexions de X Window et des fichiers de configuration installés par l'administrateur système.

6.6 Système "*Network Information Service*" (NIS)

Le système NIS (anciennement appelé "*Yellow Pages*") permet de concentrer sur une seule machine la gestion de certains fichiers utilisés pour l'administration du système Unix.

Il permet en particulier de gérer les fichiers qui contiennent les informations sur les utilisateurs (noms, mot de passe, répertoire HOME, programme de démarrage) et sur les noms et numéros IP des machines du réseau local.

```
ypcat passwd
```

affiche l'ensemble du fichier des mots de passe (si l'administrateur système l'a autorisé).

```
ypcat hosts
```

affiche l'ensemble des noms et des numéros IP des machines du réseau local.

6.7 ESPACE DISQUE OCCUPÉ

```
ypcat -k network
```

affiche l'ensemble des "*netgroups*" (voir cours sur les réseaux). L'option "**-k**" permet d'avoir les noms des groupes et pas seulement les noms des membres.

```
ypwhich -x
```

affiche les noms possibles pour les différents types de fichiers que le système NIS peut gérer.

6.7 Espace disque occupé

6.7.1 Place occupée par la branche d'un répertoire (du)

```
du [options...] [fichiers...]
```

affiche le nombre de blocs occupés par les *fichiers* indiqués (répertoire en cours par défaut). Si un des *fichiers* est un répertoire, c'est le nombre de blocs occupés par toute l'arborescence placée sous le répertoire qui est affichée.

Sous Unix Système V ou OSF les blocs ont une taille de 512 octets, sous Unix BSD ils ont une taille de 1 Ko. Consultez le manuel en ligne de *du* pour en savoir plus.

-s affiche seulement le nombre total de blocs

-a affiche la taille de tous les fichiers et répertoires

Exemple 6.2

```
du -s $HOME
```

6.7.2 Place libre d'un système de fichiers (df)

```
df [système-fichiers] [fichier]
```

affiche le nombre de blocs de 512 octets disponibles pour le système de fichiers spécifié. Le système de fichiers peut être spécifié par le nom du périphérique (*/dev/...*) ou par un fichier quelconque du système monté.

"df" (sans argument) affiche les renseignements sur tous les systèmes montés (voir 1.6.1).

Exemples 6.3

(a) **df /dev/rz3a**

affiche les renseignements sur le système de fichier du système de fichier lié au nom de fichier spécial */dev/rz3a* (une partition d'un disque).

(b) **df .**

affiche les renseignements sur le système de fichiers sur lequel se trouve le répertoire courant.

6.8 Nom du terminal utilisé (tty)

```
tty
```

affiche le nom du terminal associé à l'entrée standard du processus courant. Le code retour

(voir 14.1.6) est 0 si l'entrée standard est un terminal et 1 sinon.

Chapitre 7

Commandes pour la gestion des processus

Pour la gestion des processus on se reportera aussi aux facilités offertes par les shells *ksh* et *zsh* décrites en 14.5.1 et à la section 14.5 sur les processus lancés en arrière-plan.

7.1 Processus en cours d'exécution (ps)

```
ps [options...]
```

affiche des renseignements sur les processus en cours d'exécution et associés au terminal. Le premier chiffre est le numéro de processus, terminal qui a lancé le processus, temps d'exécution du processus, et enfin le nom de la commande qui a lancé le processus.

Si aucune option n'est spécifiée, ps donne les informations sur les processus associés au terminal de l'utilisateur qui a lancé la commande.

Cette commande offre de nombreuses options qui varient suivant les systèmes Unix. Il existe au moins trois versions différentes : BSD, OSF et Système V (ou Solaris). Voici les principales options offertes par la version Unix BSD :

- a** tous les processus lancés depuis un terminal par tous les utilisateurs,
- x** tous les processus de l'utilisateur, même ceux qui n'ont pas été lancés depuis un terminal
- l** donne des renseignements supplémentaires, en particulier le pid du processus père (PPID) de chacun des processus,
- e** affiche les valeurs des variables d'environnement,
- w** affiche en 132 colonnes (80 colonnes par défaut),
- ww** affiche autant de caractères qu'il faut pour avoir tous les renseignements sur chaque processus (utile avec l'option -e).

Exemple 7.1

Pour afficher des informations détaillées sur tous les processus sans exception on doit lancer dans les différentes versions :

en BSD : **ps -axlww**

en OSF : `ps -Al ww`
 en Système V : `ps -Afl`

7.2 Supprimer un processus en cours d'exécution (kill)

On peut supprimer un processus lancé en avant-plan en tapant la touche d'annulation (le plus souvent Ctrl-C) dans la fenêtre dans laquelle ce processus a été lancé. Pour les processus lancés en arrière-plan (voir 14.5), cette touche d'annulation n'a aucun effet. Pour les supprimer il faut utiliser la commande *kill*.

```
kill [-signal] pid
```

envoie le signal indiqué au processus dont *pid* est le numéro de processus. *signal* est un numéro de signal compris entre 0 et 15 (15 par défaut) ou le symbole d'un signal : HUP (1), INT (2), QUIT (3), KILL (9), TERM (15), etc... (taper "**kill -l**" pour avoir la liste complète des noms des signaux.)

Exemples 7.2

- (a) `kill 128`
- (b) `kill -9 128`
- (c) `kill %2`
 tue le processus lancé en arrière-plan qui a le numéro 2 pour le shell dans lequel la commande kill a été tapée.
- (d) `kill -9 -1`
 tue tous les processus lancés par l'utilisateur qui a tapé le kill. À exécuter en cas d'anomalie et d'urgence. Par exemple quand des nouveaux processus sont engendrés à grande vitesse par une erreur de programmation.

Les super-utilisateurs peuvent tuer tous les processus. L'utilisateur normal ne peut tuer que ses propres processus.

Remarque 7.1

Le signal 15 (envoyé par défaut) peut être intercepté par le processus à qui il est destiné. Ce processus peut avoir l'amabilité de laisser la place nette avant de mourir (par exemple, il peut effacer les fichiers temporaires qu'il a créés). Il peut aussi refuser de mourir et continuer de s'exécuter normalement.

Si on veut être certain de tuer un processus, il faut lui envoyer le signal 9 qui ne peut être intercepté.

Pour laisser une chance au processus de mourir "proprement", il est conseillé d'essayer d'abord de tuer avec le signal 15 et de n'utiliser le signal 9 que si le processus visé ne veut pas se laisser tuer.

7.3 Lancement automatique de processus à des moments donnés (at, crontab)

Si l'administrateur du système l'a autorisé (voir le manuel en ligne des commandes *at* et *crontab*), les utilisateurs peuvent programmer le lancement automatique de processus même lorsqu'il ne sont pas connectés.

at sert à lancer une commande à un moment donné. *crontab* est plus spécialement utilisé pour lancer des commandes à intervalles réguliers (chaque jour par exemple).

Sans entrer dans les détails, voici des exemples standards :

```
at -f ~/fichier 3:00 am January 24
```

lance les commandes contenues dans *~/fichier* (indiquez toujours un nom absolu) 3 heures du matin le 24 janvier.

```
at -l
```

affiche les commandes programmées.

```
crontab fichier-cron
```

programme ce qui est indiqué dans le fichier *fichier-cron*. Voici un exemple de contenu d'un tel fichier (voir le manuel en ligne pour plus d'explications) :

```
# Lance 'dumputout -f' tous les jours (sauf le lundi) à 2h du matin
```

```
# (0 minutes), tous les jours du mois, tous les mois,
```

```
# le dimanche (0) et du mardi au vendredi (2-6)
```

```
0 2 * * 0,2-6 /usr/adm/dumputout -f
```

```
crontab -l
```

affiche les commandes programmées.

```
crontab -e
```

permet de modifier les commandes programmées.

7.4 Gestion du plan d'un processus par le shell

La plupart des shells (mais pas *sh*) permettent de gérer le plan où se déroule les processus (en arrière-plan ou non). Cette gestion est étudiée en 14.5.1.

Chapitre 8

Afficher, imprimer, envoyer le contenu d'un fichier

8.1 Afficher le contenu d'un fichier, concaténer plusieurs fichiers (cat) - Notion de redirection

```
cat [-nsv] [fichiers...]
```

affiche le contenu des *fichiers*.

Si aucun fichier n'est spécifié, l'entrée standard est prise par défaut.

Options

- n numérote les lignes
- s réunit plusieurs lignes vides en une seule ligne
- v affiche les caractères non visibles

Exemples 8.1

(a) `cat /etc/passwd`

(b) `cat`
affiche (plus exactement, envoie sur la sortie standard) les lignes qui sont tapées au clavier. On termine en tapant Ctrl D qui est transformé en annonce de “fin de fichier” par le driver de terminal. C'est surtout utile avec une redirection (voir ci-dessous).

Le nom de la commande vient de *concaténer* car

```
cat fich1 fich2 >fich3
```

concatène fich1 et fich2 dans fich3. “>” est un symbole de **redirection** (la sortie standard est redirigée vers fich3 ; par défaut, la sortie standard est l'écran). Cette notion sera détaillée en 14.2.

La commande cat permet de créer des petits fichiers sans passer par un éditeur de textes (“\$” est le prompt dans l'exemple suivant) :

```
$ cat >nouveaufichier  
ligne1
```

```
ligne2
```

<-- on tape la touche [Ctrl] D
crée “nouveaufichier” avec deux lignes de texte.

8.2 Afficher les octets d'un fichier (od)

Cette commande ne sera pas étudiée ici mais il faut connaître son existence pour le cas où l'on doit examiner le contenu des fichiers qui ne contiennent pas du texte affichable.

emacs possède aussi le mode de travail “Hexl” qui permet de voir, et même de modifier les octets d'un fichier. On tape “M-x hexl-find-file” pour charger le fichier binaire. On peut aussi simplement passer en mode Hexl une fois que l'on a chargé un fichier dans un buffer (“M-x hexl-mode”).

8.3 Afficher page à page (more) - Notion de pipe

```
more [fichiers...]
```

affiche le contenu des *fichiers* page par page. Le fichier par défaut est l'entrée standard.

En bas de chaque page on dispose de plusieurs commandes :

- | | |
|------------------|---|
| h | affiche la liste des commandes disponibles |
| [Retour] | affiche une ligne de plus |
| [espace] | affiche l'écran suivant |
| i[espace] | affiche i lignes de plus |
| Ctrl B | revient à la page précédente (ne marche pas si le flot de donnée ne vient pas d'un fichier ordinaire) |
| q ou Q | sort de more |
| = | affiche le numéro de la ligne en cours |
| /texte | recherche un <i>texte</i> : affiche un nouvel écran qui commence 2 lignes avant la 1ère ligne qui contient <i>texte</i> |
| n | refait la dernière recherche |
| :n | va au fichier suivant |
| :f | affiche le nom du fichier et le numéro de la ligne |

Grâce à un *pipe* (notion étudiée en 14.3) la commande *more* peut recevoir sur son entrée standard ce qu'une autre commande a envoyé vers la sortie standard. On peut ainsi effectuer un affichage page à page de ce qu'aurait affiché cette autre commande. Par exemple :

```
ls -l /bin | more
```

8.4 Sorties sur les imprimantes

Lorsque l'utilisateur lance l'impression d'un fichier, le contenu de ce fichier est envoyé dans une file d'attente. Il sera imprimé lorsque le programme gestionnaire du spouleur (trac

duction du terme anglais *spool* qui est d'ailleurs le plus souvent utilisé) le sélectionnera. Ce système de spool permet de redonner la main tout de suite à l'utilisateur sans attendre la fin de l'impression du contenu du fichier.

Il existe deux systèmes d'impression : *lp* (Unix Système V) et *lpr* (Unix BSD). Le système décrit ici est le système *lpr*. Avec le système *lp*, la commande *lp* remplace la commande *lpr* et la commande *lpstat* donne des informations sur le système d'impression, en particulier sur les noms des imprimantes. Les systèmes Unix actuels disposent des deux systèmes d'impression et des passerelles sont souvent installées par l'administrateur pour laisser le choix à l'utilisateur : un seul des deux systèmes fonctionne mais les commandes de l'autre système sont disponibles (elles font appel au seul système installé sans que l'utilisateur s'en aperçoive).

8.4.1 Informations sur le système d'impression (lpstat, printcap)

En Unix BSD, le fichier `/etc/printcap` contient les noms et descriptifs des imprimantes du système. Sous Unix Système V et OSF, la commande *lpstat* permet d'avoir des informations sur le système d'impression.

```
lpstat [options]
```

Options

-d donne l'imprimante par défaut
-p statut des imprimantes et de la gestion des files d'attente associées
-r état du démon d'impression
-s un résumé des précédentes options
-t tout ce que vous avez toujours voulu savoir sur le système d'impression
-v noms des imprimantes et des noms de fichiers spéciaux ou de machines distantes associées

Exemple 8.2

Certaines de ces options peuvent être suivies de listes d'imprimantes, d'utilisateurs ou d'identificateurs de requêtes d'impression.

```
lpstat -p lp17,lp30
```

8.4.2 Lancement d'une requête d'impression (lpr)

```
lpr [options] [fichiers...]
```

place une liste de fichiers dans la file d'attente d'une imprimante. Un saut de page est envoyé après chaque fichier.

Si la liste de fichiers est vide, c'est l'entrée standard qui est placée en file d'attente (ce qui permet d'utiliser *lpr* à la fin d'un pipe).

Dans tout système, l'administrateur donne une imprimante par défaut vers laquelle les requêtes sont dirigées (imprimante de nom "lp"). La variable PRINTER peut contenir le nom d'une autre imprimante. On peut aussi utiliser l'option -P.

Les options sont nombreuses et peuvent varier suivant la version du système.

8.4. SORTIES SUR LES IMPRIMANTES

Options

-Pimprimante dirige l'impression vers l'imprimante spécifiée
-h supprime la bannière d'impression (première page où sont imprimés le nom de l'utilisateur qui a lancé l'impression et diverses autres informations)

Exemples 8.3

(a) `lpr -h -Plp3 fichier`

(b) `ls -l | lpr -h`

8.4.3 Informations sur les requêtes d'impression (lpq)

```
lpq [-Pimprimante] [utilisateur]
```

affiche des renseignements (utilisateur, position dans la file d'attente, nom du fichier à imprimer, identificateur de la requête, taille de ce qui sera imprimé) sur les requêtes d'impressions actuellement en attente d'être imprimées.

Options

-Pimprimante

limitent les informations aux requêtes envoyées sur l'imprimante

utilisateur

limitent les informations aux requêtes lancées par utilisateur.

Exemple 8.4

```
lpq -Plp3 jean
```

donne les informations sur les requêtes envoyées par jean sur l'imprimante lp3.

8.4.4 Suppressions de requêtes d'impression (lprm, cancel)

```
lprm [-Pimprimante] [-] [identificateur-requete]
```

permet à un utilisateur de supprimer une requête d'impression qu'il avait précédemment lancée (si l'impression est déjà commencée, elle est interrompue). On ne peut supprimer les requêtes d'un autre utilisateur (sauf si l'on est le super-utilisateur). Si aucune option n'est spécifiée, la requête en cours d'impression est supprimée sur l'imprimante par défaut si elle appartient à l'utilisateur qui a lancé la commande *lprm*.

Options

-Pimprimante

supprime toutes les requêtes que l'utilisateur a lancées sur imprimante

-

supprime toutes les requêtes que l'utilisateur a lancées

identificateur-requete

supprime la requête définie par son identificateur (on peut faire afficher l'identificateur par la commande *lpq* étudiée en 8.4.3).

Exemples 8.5

(a) `lprm -Plp1 26`

(b) `lprm -`

Si la commande *lprm* semble ne pas fonctionner, on peut essayer la commande *cancel*. Par exemple, pour supprimer les requêtes que l'utilisateur toto a envoyées sur l'imprimante lp0, on tapera :

```
cancel -u toto lp0
```

8.5 Envoyer le contenu d'un fichier (mail)

Le plus simple pour envoyer par courrier électronique un fichier est d'utiliser la commande *mail* avec une redirection de l'entrée standard (voir 14.2.4) :

```
mail adresse <fichier
```

envoie le contenu de *fichier* à l'utilisateur désigné par *adresse*.

Le fichier doit être un fichier texte. Si on veut envoyer un fichier binaire, il faut le traiter auparavant par *uuencode* (voir cours sur les réseaux).

Exemple 8.6

```
mail -s "Fichier fich" toto@machine.unice.fr < rep/fich
```

L'option “-s” permet de donner un sujet au message ; elle n'est pas disponible sur toutes les versions de la commande.

Nous ne décrivons pas davantage la commande *mail* puisque nous utiliserons d'autres commandes (*netscape*, *xmh* ou *emacs*) pour envoyer et recevoir du courrier électronique.

8.6 Mise en page (pr)

```
pr [options] [fichiers...]
```

met en forme les *fichiers* (longueur de ligne, longueur de page, en-tête, ...) et envoie le résultat vers la sortie standard. *pr* prend l'entrée standard par défaut si aucun fichier n'est indiqué.

pr possède de nombreuses options que nous ne décrivons pas ici.

8.7 Passer en Postscript (a2ps)

```
a2ps [options-globales] [[options-partic.] fichiers]...
```

passse le contenu d'un ou plusieurs fichiers (entrée standard par défaut) en Postscript.

C'est une commande très utile pour imprimer le contenu de fichiers sur des imprimantes Postscript. En particulier, les programmes informatiques sont souvent traités d'une façon particulière par *a2ps* qui imprimera par exemple tous les mots-clés du langage en caractères gras.

Attention, le fonctionnement standard dépend des versions installées, en particulier pour l'endroit où est envoyé le Postscript généré.

De plus, les noms des options peuvent changer avec les versions de *a2ps*. Heureusement l'une des options que l'on retrouve dans toutes les versions est l'option “-h” qui donne une liste des options disponibles.

Les options globales permettent d'indiquer si le Postscript généré est envoyé sur la sortie standard ou sur une certaine imprimante, s'il faut imprimer les fichiers binaires, les pages auront une en-tête, etc.

Les options particulières à chaque fichier (ou groupe de noms de fichiers qui suivent les options) permettent d'indiquer si on veut une ou deux pages par feuille physique de papier, si on veut le mode de positionnement “portrait” ou “paysage” (sur la largeur de la page *landscape* en anglais), etc. Il existe de très nombreuses options, souvent très utiles que l'on aura intérêt à lire dans le manuel en ligne.

Exemple 8.7

Sur la version de *a2ps* installée aujourd'hui sur les stations Sun de l'université de Nice, la commande suivante imprime le fichier truc.java sur l'imprimante par défaut deux pages par feuille en mode paysage :

```
a2ps truc.java | lpr
```

Pour imprimer une page par feuille en mode “portrait” :

```
a2ps -P -1 truc.java | lpr
```

Chapitre 9

Gestion des fichiers

Ce chapitre traite des commandes qui travaillent sur des fichiers ordinaires en les considérant comme un tout, sans faire d'opérations sur le contenu des fichiers.

9.1 Copier des fichiers (cp)

```
cp [-ip] fichier1 fichier2
```

copie le fichier ordinaire *fichier1* sur le fichier ordinaire *fichier2*.

L'utilisateur doit avoir l'autorisation de lecture sur le fichier *fichier1*.

Si *fichier2* existe, l'utilisateur doit avoir l'autorisation d'écriture dans ce fichier. Dans ce cas, le contenu de *fichier2* est écrasé par celui de *fichier1*. *fichier2* garde son propriétaire, son groupe et ses autorisations.

Si *fichier2* n'existe pas, l'utilisateur doit avoir l'autorisation d'écriture sur le répertoire de *fichier2*. Dans ce cas, *fichier2* est créé et son propriétaire et son groupe sont ceux de l'utilisateur et les autorisations sont celles de *fichier1* (mais tient compte de la valeur du masque de umask ; voir 5.2).

Options

-i fait afficher un message de demande de confirmation si *fichier2* existe déjà
-p conserve les dates de modification et d'accès, le propriétaire et le groupe, le mode d'accès de *fichier1* pour *fichier2* (ne tient pas compte du masque pour le mode d'accès ; voir 5.2)

```
cp [-ip] fichiers... répertoire
```

copie les fichiers dans le répertoire.

Exemples 9.1

(a) `cp fich* jean/courrier`

(b) `cp fichier1 fichier2 jean`

Remarque 9.1

Si un des *fichiers* est un répertoire, *cp* envoie un message d'erreur mais les autres fichiers sont copiés (voir 10.4 pour la copie de répertoires).

9.2 Liens avec même numéro de i-node (ln)

Un répertoire contient des *liens* qui référencent les fichiers placés “sous” le répertoire. Un fichier peut avoir plusieurs liens qui le référence. On peut ajouter un lien vers un fichier dans un autre répertoire que le répertoire de création du fichier en utilisant la commande *ln*.

Un lien permet de donner plusieurs noms à un même fichier physique. La création d'un lien permet, par exemple, de partager une commande entre plusieurs utilisateurs sans dupliquer son contenu.

```
ln [-f] fichier1 [fichier2]
```

crée un lien *fichier2*: *fichier2* sera un fichier qui aura le même i-node que *fichier1*.

fichier1 doit déjà exister et ne pas être un répertoire. Les deux fichiers doivent appartenir à un même système de fichiers.

L'utilisateur doit avoir l'autorisation d'écriture sur le répertoire de *fichier2*. L'autorisation d'écriture sur le fichier *fichier2* n'est pas nécessaire (comme pour *rm* ; voir 9.4).

fichier2 est optionnel. S'il n'est pas indiqué, le lien se fait sur le fichier situé dans le répertoire courant de même nom terminal que *fichier1*.

L'option “**-f**” permet d'écraser *fichier2* s'il existe déjà.

```
ln [-f] fichiers... répertoire
```

crée des liens entre les fichiers et les fichiers situés sous le *répertoire* et de même nom terminal que les *fichiers*. Les *fichiers* doivent appartenir à un même système de fichiers que le *répertoire*. Les *fichiers* ne peuvent être des répertoires.

Remarques 9.2

- Quand un lien a été créé entre deux fichiers, les deux fichiers jouent ensuite un rôle symétrique et il n'y a pas lieu de distinguer un fichier originel parmi tous ces fichiers.
- Tous les fichiers liés ont le même i-node. Ceci implique que lorsque l'on modifie le contenu ou les autorisations d'un fichier, tous les fichiers qui lui sont liés sont modifiés de même. Le propriétaire, le groupe et les dates de dernière modification et de création sont les mêmes pour tous les fichiers liés correspondant à un même fichier physique.
- La commande “**ls -l**” renverra un nombre de liens supérieur à 1 pour les fichiers ainsi liés.
- Si on supprime un des deux fichiers par la commande *rm*, l'autre continuera à exister avec un nombre de lien affiché par “**ls -l**” diminué de 1.

Exemples 9.2

(a) `ln rep1/fichier rep2`

(b) `ln rep1/fichier rep2/fichier2`

9.3 Liens symboliques (ln -s)

Les liens étudiés dans la section précédente ont quelques limitations. En particulier, les deux fichiers liés doivent être placés dans un même système de fichiers et les fichiers auxquels on ajoute un lien (ceux que l'on référence) ne peuvent être des répertoires. Les liens symboliques offrent plus de souplesse.

Un lien symbolique est un fichier de type spécial qui contient le nom d'un autre fichier auquel il est lié. Toutes les commandes usuelles qui contiendront le nom du lien symbolique, travailleront en fait avec le fichier dont le nom est contenu dans le lien symbolique.

```
ln -s fichier1 [fichier2]
```

crée un lien symbolique *fichier2* qui se référera à *fichier1*. *fichier1* et *fichier2* peuvent être des répertoires et ils peuvent appartenir à deux systèmes de fichiers différents. *fichier1* peut même être supprimé ensuite (et dans ce cas, toute référence à *fichier2* provoquera une erreur) !

fichier1 peut être un répertoire.

fichier2 est optionnel. S'il n'est pas indiqué, le lien se fait avec le fichier situé dans le répertoire courant de même nom terminal que *fichier1*.

On peut aussi indiquer un répertoire comme but (comme pour les commandes *cp* ou *mv*) :

```
ln -s fichiers... repertoire
```

Exemple 9.3

```
ln -s /usr/new/emacs/emacs bin
crée un lien symbolique bin/emacs qui "pointe" sur /usr/new/emacs/emacs.
```

Remarques 9.3

- (a) "**ls -l fichier**" d'un fichier (ou répertoire) qui est un lien symbolique, affiche le nom du fichier sur lequel le fichier est lié. Si on veut les renseignements sur le fichier pointé (ou les sous-fichiers du répertoire pointé), il faut utiliser l'option **-L** de *ls*. Par exemple, "**ls -lL fichier**"
- (b) les autorisations sur un lien symbolique n'ont aucune signification : les droits réels sont ceux du fichier pointé.
- (c) on fera attention de donner un nom absolu pour *fichier1*, ou un nom relatif par rapport au répertoire de *fichier2* pour que le fichier pointé soit retrouvé lorsque le système utilisera le lien symbolique.

9.4 Supprimer des fichiers (rm)

```
rm [options...] fichiers...
```

enlève le fichier de son répertoire.

9.5 DÉPLACER, RENOMMER DES FICHIERS (mv)

Cette commande nécessite l'autorisation d'écriture sur le répertoire du fichier mais pas la lecture ou l'écriture sur le fichier lui-même (mais si l'utilisateur n'a pas l'autorisation d'écriture sur le fichier, une confirmation est demandée).

Options

- f** pas de demande de confirmation quand on n'a pas l'autorisation d'écriture sur le fichier, renvoie un code retour 0 (tout s'est bien passé) même si un des *fichiers* n'existe pas
- r** détruit tout le contenu d'un répertoire et le répertoire lui-même
- i** demande de confirmation pour chaque fichier (ou répertoire avec l'option **-r**)

Exemple 9.4

```
rm -fr l*
```

Attention à ne pas glisser par mégarde un espace entre le "l" et le "*". On se méfie aussi des claviers dont la touche "l" ne fonctionne pas bien ! Il est prudent de lire sur l'écran la commande tapée avant de la lancer.

9.5 Déplacer, renommer des fichiers (mv)

```
mv fich1 fich2
```

renomme *fich1* en *fich2*. Le fichier peut changer de répertoire à cette occasion.

```
mv fichiers... repertoire
```

les *fichiers* passent sous *repertoire*. Ils gardent le même nom terminal.

Exemples 9.5

(a) **mv fichier1 rep/fichier2**

(b) **mv jean/fich1 jean/fich2 pierre**

9.6 Sauvegarder sur les lecteurs de disquettes des stations Sun

Certaines stations Sun de l'université de Nice ont des lecteurs de disquettes compatibles avec les lecteurs installés sur les ordinateurs compatibles PC. Les utilisateurs peuvent ainsi recopier sur des disquettes des programmes récupérés sur le réseau Internet ou leurs fichiers de travail pour les emmener sur leur PC personnel.

Les sauvegardes sont effectuées régulièrement par l'administrateur du système sur bandes ou autres supports. Pour les fichiers importants, deux précautions valent mieux qu'une et on peut faire ses propres sauvegardes si on a accès à des stations Sun.

Les commandes qui permettent de travailler avec les disquettes sont les commandes *mcopy*, *mdir*, etc. (chercher toutes les commandes dont le nom commence par "m" dans le répertoire contenant la commande *mcopy*). De plus, la commande *eject* permet d'éjecter la disquette du lecteur. On pourra consulter le manuel en ligne pour plus de détails.

9.7 Compression et décompression (zip, gzip, compress)

Ces trois commandes permettent de compresser les fichiers. On peut ainsi archiver des fichiers non utilisés actuellement en sauvegardant de la place. La commande *zip* se trouve sur PC MS-DOS et sur Unix.

Pour compresser des arborescences (tout le contenu d'un répertoire et de ses sous-répertoires) on pourra utiliser la commande *tar* (voir 10.6).

Consultez les manuels en ligne de ces commandes pour plus de précisions.

Chapitre 10

Travail sur les répertoires

Les commandes principales de gestion des fichiers que l'on a vu au chapitre précédent permettent de travailler sur les répertoires. Ce chapitre indique seulement les options et fonctionnalités liées aux répertoires. La plupart des options déjà vues (par exemple les options *-i* de *cp* ou *-f* de *rm*) sont toujours valables pour les répertoires.

10.1 Créer un répertoire (mkdir)

```
mkdir [-p] répertoires...
```

crée les *répertoires* indiqués.

On doit avoir l'autorisation d'écriture dans le répertoire père.

En l'absence de masque donné par la commande *umask* (voir 5.2), les autorisations sur le nouveau répertoire sont 777.

L'option “*-p*” crée les répertoires intermédiaires si besoin est.

10.2 Supprimer un répertoire (rmdir, rm -r)

```
rmdir répertoires...
```

supprime les *répertoires* indiqués. Les répertoires doivent être vides.

Un autre moyen de supprimer un répertoire :

```
rm -r répertoires...
```

supprime les *répertoires* et toute l'arborescence dont ils sont la “racine”.

ATTENTION ! *rm -r ** supprime tout le répertoire de travail.

10.3 Changer le nom d'un répertoire (mv)

```
mv répertoire1 répertoire2
```


Autorisé seulement si les 2 répertoires ont le même parent en Unix Système V. Unix BSD l'autorise si les deux répertoires sont dans le même système de fichiers.

Exemple 10.1

```
mv /users/students/jean /users/students/jp
```

10.4 Copier l'arborescence d'un répertoire (cp -r)

```
cp -r répertoire1 répertoire2
```

copie toute l'arborescence de *répertoire1* sous *répertoire2*.

L'endroit où seront copiés les fichiers dépend de l'existence du répertoire *répertoire2*. Si celui-ci n'existe pas, il sera créé et il correspondra à *répertoire1*. Sinon, les nouveaux fichiers se retrouveront sous le répertoire situé dans le répertoire *répertoire2* et de même nom terminal que le nom terminal de *répertoire1*.

Exemples 10.2

- (a) `cp -r ~toto/rep .`
crée (s'il n'existe pas déjà) un répertoire “rep” sous le répertoire courant et copie les fichiers de l'utilisateur “toto” sous ce répertoire
- (b) `cp -r ~toto/rep/. .`
copie les fichiers de “rep” directement sous le répertoire courant

Remarque 10.1

La commande suivante copie les fichiers placés *directement sous* un répertoire **rep1** dans un répertoire **rep2**.

```
cp rep1/* rep1/. * rep2
```

Les sous-répertoires ne seront pas copiés et engendreront des messages d'erreurs.

10.5 Lien symbolique (ln -s)

Un lien symbolique peut pointer un répertoire. Ceci permet d'obtenir une configuration attendue par un logiciel. Par exemple, beaucoup de logiciels du domaine public travaillent par défaut avec des sous-répertoire de `/usr/local`. Un lien sur ces répertoires peut éviter de modifier les fichiers de configuration des logiciels lorsque c'est impossible ou trop complexe.

Exemple 10.3

Supposons que le système de fichier qui contient le répertoire `/usr/local` n'a pas assez de place pour accueillir dans `/usr/local/rep1` tous les fichiers nécessaires au fonctionnement d'un logiciel. Si le paramétrage du logiciel pour lui indiquer de chercher les fichiers à un autre endroit que `/usr/local/rep1` est impossible ou complexe, on peut installer les fichiers dans le répertoire `/usr/local2/rep1` et taper :

10.6. PLIAGE DE RÉPERTOIRES EN UNE SEUL FICHIER (TAR, CPIO)

```
ln -s /usr/local2/rep1 /usr/local/rep1
```

Si le répertoire `/usr/local/rep1` existe déjà, on créera des liens symboliques sous le répertoire `/usr/local/rep1` pour tous les fichiers de `/usr/local2/rep1` qui sont concernés ; par exemple, pour `fichier1` :

```
ln -s /usr/local2/rep1/fichier1 /usr/local/rep1
```

10.6 Pliage de répertoires en une seul fichier (tar, cpio)

La commande *tar* peut “plier” un répertoire en un seul fichier. Tous les fichiers du répertoire sont alors réunis en ce fichier. Le transfert des fichiers sur le réseau et leur archivage en est facilité, surtout si on compacte le fichier résultat.

Cette commande possède de nombreuses options. On ne donne ici que les options principales sur des exemples simples.

```
tar cvf fichier.tar rep
```

crée le fichier `fichier.tar` qui contiendra tous les fichiers des fichiers indiqués. Si un fichier indiqué est un répertoire (comme ici **rep**), toute l'arborescence du répertoire est pliée dans le fichier dont le nom complète l'option “f” (l'option “v” est l'option “verbeuse”).

Il est fortement conseillé d'utiliser un nom relatif pour les fichiers pliés car on peut ainsi récupérer les fichiers (par “tar x”) à un autre endroit.

```
tar tvf fichier.tar
```

liste les noms des fichiers pliés dans le fichier `fichier.tar`.

```
tar xvf fichier.tar
```

récupère tous les fichiers pliés dans `fichier.tar`.

```
tar xvf fichier.tar f1 f2
```

récupère tous les fichiers pliés dans `fichier.tar` qui ont pour nom `f1` ou `f2`.

La commande *cpio* permet aussi l'archivage de fichiers. Elle ne sera pas étudiée ici.

Chapitre 11

Expressions régulières

Ces expressions sont utilisées par plusieurs commandes ; en particulier par *more*, *emacs*, *grep*, *ed*, *sed*, *awk*, *perl*¹. Ce sont des chaînes de caractères qui représentent des modèles pour des chaînes de caractères. Certains caractères ont des significations particulières.

Toutes les commandes qui utilisent les expressions régulières ne reconnaissent pas toutes les expressions. Les commandes *grep*, *more* et *emacs* travaillent avec le noyau minimum d'expressions régulières qui sont présentées dans ce chapitre. Il existe d'autres types d'expressions régulières. Pour connaître exactement les expressions reconnues par une commande, il faut se référer au manuel de référence, ou faire des tests... De plus, les différentes versions d'une commande (par exemple la commande *grep*) ne reconnaissent pas le même ensemble d'expressions régulières.

Attention à ne pas confondre les expressions régulières qui ne sont reconnues que par certaines commandes avec les expressions qui contiennent des caractères spéciaux interprétés par le *shell* (voir 4.5) et qui sont donc reconnues dans toutes les lignes de commandes.

11.1 Expressions régulières représentant un seul caractère

Un caractère ordinaire se désigne par lui-même. Les expressions régulières peuvent aussi contenir des caractères spéciaux. Les expressions régulières suivantes désignent un seul caractère ; elles sont reconnues par toutes les commandes :

<code>\</code> <code>^</code> <code>\$</code> <code>.</code>	suivi d'un autre caractère désigne ce caractère. Il permet d'enlever le sens spécial des caractères comme <code>.</code> , <code>*</code> , <code>[</code> et <code>\</code> . placé en début d'expression, désigne un début de ligne. placé en fin d'expression, désigne une fin de ligne. (le point) désigne n'importe quel caractère excepté le passage à la ligne (équivalent de <code>?</code> du langage de commande).
---	---

¹. cette commande n'est pas étudiée dans ce cours

11.2. EXPRESSIONS RÉGULIÈRES REPRÉSENTANT UN ENSEMBLE DE CARACTÈRES

[chaîne]

désigne n'importe quel caractère de la chaîne. On peut désigner plusieurs caractères contigus à l'aide de `"`. Si le premier caractère de la chaîne est `^`, l'expression désigne n'importe quel caractère qui n'est pas dans la chaîne.

Exemples 11.1

- (a) `^ab`
désigne la chaîne "ab" placée en début de ligne.
- (b) `[0-9]`
désigne un chiffre quelconque.
- (c) `[adh-1A-Z]`
désigne a ou d ou une lettre minuscule comprise entre h et l, ou une lettre majuscule comprise entre A et Z.
- (d) `[~0-9]`
désigne n'importe quel caractère qui n'est pas un chiffre.
- (e) `^\.`
désigne une ligne qui commence par un point.
- (f) `[-d]`
désigne un "-" ou un "d".

11.2 Expressions régulières représentant un ensemble de caractères d'un seul type

À partir des expressions régulières correspondant à un seul caractère (et seulement à partir de ces expressions), on peut construire les expressions régulières suivantes :

- *** une expression ci-dessus suivie du caractère `"*"` désigne 0 ou plus occurrences des caractères désignés par l'expression. S'il y a plusieurs possibilités, l'expression désigne la plus longue chaîne à partir de la gauche qui permette de trouver une chaîne correspondant à l'expression régulière. La signification de `*` diffère donc de la signification habituelle lors de l'interprétation des commandes par le *shell*.
- + et ?** certaines commandes, (mais pas toutes les versions de *grep*) donnent un sens particulier aux caractères `"+"` et `"?"` :
`?` désigne 0 ou 1 occurrence des caractères désignés par l'expression qui précède.
`+` désigne 1 ou plusieurs occurrences des caractères désignés par l'expression qui précède.

Pour les commandes qui ne reconnaissent pas le `"+"`, on peut le simuler ; par exemple, `"xx*"` peut remplacer `"x+"`.

`\{ et \}` certaines commandes (mais pas toutes les versions de *grep*) permettent de donner encore plus de précision sur le nombre d’occurrences : on peut faire suivre une expression régulière par (*m* et *n* sont des entiers compris entre 0 et 256) :

`\{m\}` pour indiquer exactement *m* occurrences

`\{m,\}` pour indiquer au moins *m* occurrences

`\{m,n\}` pour indiquer entre *m* et *n* occurrences (bornes comprises)

Exemples 11.2

- (a) `^[a-zA-Z]*`
désigne la plus longue chaîne de lettres à partir du début de la ligne ; si la ligne contient “Bonjour Monsieur”, c’est “Bonjour” qui est désigné.
- (b) `[0-9]\{3\}`
désigne un nombre de 3 chiffres.

11.3 Autres expressions régulières

On peut construire des expressions régulières à partir d’autres expressions régulières :

concaténation

la concaténation d’expressions régulières désigne la concaténation des chaînes désignées par chacune des expressions régulières.

Exemple 11.3

`[0-9]\{3\}A.*T$`

désigne une chaîne qui commence par 3 chiffres suivis de la lettre A et qui se termine par la lettre T à la fin de la ligne.

`\(...\) et \n`

Les commandes *grep* (cela dépend des versions) et *awk* permettent de désigner des sous-chaînes de caractères à l’aide des deux types d’expressions suivants :

`\(expression\)` désigne la même chose que l’*expression* elle-même (voir utilité ci-après)

`\n` où *n* est un entier désigne la chaîne de caractères qui correspond à la *n^{ième}* expression entourée précédemment par `\(` et `\)`.

Exemples 11.4

- (a) `^\(.*\)1$`
désigne une ligne formée d’une chaîne répétée 2 fois.
- (b) `who | grep_^\([a-z]\)^\[^\]*1_`
affiche des informations sur les utilisateurs connectés à la machine locale et dont le nom de login commence et se termine par la même lettre minuscule (“`_`” désigne un espace).

Des possibilités supplémentaires sont utilisées par certaines commandes et en particulier par la commande *awk* (mais généralement pas par *ed* ou *grep*) :

expression1|expression2

désigne une chaîne correspondant à la 1^{ère} ou à la 2^{ème} expression.

(...) les parenthèses permettent de modifier l’ordre de priorité des opérateurs. L’ordre de priorité est `[et]`, puis `*`, `+` et `?`, puis la concaténation et enfin `|`.

Chapitre 12

Éditeur de texte (emacs, xemacs)

Pour écrire des textes (programmes ou données) sous Unix, il faut disposer d'un programme qui permette d'entrer du texte et de le modifier sans être obligé de tout retaper. Les programmes qui font ce genre de travail s'appellent des éditeurs de textes. Ils possèdent de nombreuses commandes qui facilitent la tâche de l'utilisateur : recherche de chaînes de caractères dans un texte, remplacement automatique de chaînes par une autre, copie, suppression de partie de texte, etc.

L'éditeur emacs est un éditeur pleine page très puissant. L'éditeur pleine page le plus commun sous Unix est *vi* mais il est moins agréable à utiliser que emacs, moins puissant et moins souple.

Une caractéristique essentielle de emacs est qu'il est possible (et pas trop complexe) de personnaliser son fonctionnement en affectant des valeurs à des variables d'environnement ou même en programmant des nouvelles fonctionnalités.

Nous ne donnerons ici que les commandes essentielles de *emacs*, suffisantes pour commencer à écrire des programmes et des textes simples sous Unix. La version décrite est la version 19 avec quelques ajouts pour la version 20. Cette dernière version offre de nombreuses facilités pour travailler avec des langues non européennes et pour configurer plus simplement *emacs*.

xemacs est une version d'*emacs* plus graphique mais moins standard. Il possède beaucoup de points en commun avec *emacs* mais certains points sont incompatibles. Nous ne l'étudierons pas dans ce cours mais *xemacs* est disponible sur la plupart des machines de l'université de Nice.

12.1 Entrée et sortie

12.1.1 Lancer emacs

```
emacs [-q] [fichier] &
```

lance la version X Window de emacs. Comme tous les clients X, il faut lancer emacs en arrière-plan pour garder la possibilité de travailler dans la fenêtre X d'origine. Dans un

12.2. CONCEPTS DE BASE

environnement X Window, emacs est lancé dans sa propre fenêtre X.

Le noyau d'*emacs* charge en mémoire de nombreux modules écrits en *elisp* pour étendre ses fonctionnalités. Il faut indiquer à l'éditeur où aller chercher ces modules. Si on utilise un seul éditeur de la famille *emacs*, on peut utiliser la variable d'environnement `EMACSLoadPath`. Pour cela, on ajoutera, par exemple, les lignes suivantes dans le fichier `.zshrc` (voir 15.3.2) :

```
export EMACSDIR=/usr/local/emacs-19.31
export EMACSLoadPath=$HOME/emacs:$EMACSDIR/site-lisp:$EMACSDIR/lisp
```

Attention, *emacs* et *xemacs* utilisent des variantes différentes de certains modules. On évitera d'utiliser la variable `EMACSLoadPath` si on utilise les deux éditeurs. Dans ce cas, le langage *elisp* permet de donner des valeurs différentes selon le type, et même la version de l'éditeur qui lit le fichier `.emacs`. L'écriture d'un fichier `.emacs` ne fait pas partie de ce cours ; récupérez des exemples déjà écrits de fichiers `.emacs` et/ou lisez le manuel du langage si vous êtes intéressés.

Le fichier `.emacs` du répertoire HOME de l'utilisateur est exécuté automatiquement au démarrage d'*emacs*. Si on ne veut pas qu'il s'exécute (par exemple si ce fichier contient des erreurs), on lance *emacs* avec l'option `-q`.

12.1.2 Sortir de emacs

Sous X Window, le menu "File" permet de sortir de emacs. `C-x C-c` permet de sauvegarder (après demande de confirmation) les buffers avant de sortir.

12.2 Concepts de base

12.2.1 Description d'une fenêtre emacs

Une fenêtre affichée par emacs est composée de :

- une barre de menus (seulement pour les versions sous graphiques en haut de la fenêtre)
- la fenêtre de travail proprement dite où s'affiche le buffer courant, sous la barre de menus,
- la ligne d'information, située sous la fenêtre de travail,
- le "mini-buffer" se trouve en bas de la fenêtre ; il contient en général une seule ligne.

Dans la version X Window, un ascenseur borde la droite de la fenêtre de travail.

Menus de la version graphique (X Window)

- Si on travaille dans l'environnement graphique (X Window, Windows 95 ou autre), le haut de l'écran est occupé par une ligne de noms de menus :
- Buffers permet de choisir son buffer de travail
 - File permet de charger, sauvegarder un fichier, ouvrir une nouvelle fenêtre, imprimer ou supprimer un buffer, sortir de emacs
 - Edit permet de copier, couper, coller, défaire l'effet d'une commande, remplir avec le plus de mots possible les lignes d'une région ("Fill")
 - Help permet d'obtenir de l'aide sur emacs ou une commande Unix

De nombreux modes de travail (voir 12.2.5) modifient ces menus ou ajoutent de nouvelles entrées.

Ligne d'information sur le mode de travail

La dernière ligne en bas de l'écran contient quelques indications utiles à l'utilisateur :
--*-Emacs : nom-buffer (mode majeur mineur)--L106--position-----
Les “*-” indiquent que le buffer a été modifié (on peut aussi avoir %% qui indique que le buffer est “*readonly*”).
mode majeur et mode mineur sont les modes de travail (voir 12.2.5).
“L106” désigne le numéro de la ligne courante.
“position” peut être All, Bot, Top ou nn% suivant que tout le texte est affiché ou seulement la fin ou le début, ou si nn% du texte est au dessus de la première ligne affichée.

Mini-buffer

Les commandes tapées par l'utilisateur et les messages envoyés par emacs sont affichés sous la ligne d'information dans une zone appelée minibuffer.
On peut annuler une commande en cours et affichée dans ce minibuffer en tapant C-g.

12.2.2 Commandes et associations de clés

Touches spéciales

emacs fait un grand usage de deux touches spéciales notées “C-” et “M-” dans le manuel d'utilisation.
“C-” désigne la touche [Ctrl]. Taper “C-x” revient à appuyer sur la touche [Ctrl], et à appuyer sur la touche “x” tout en gardant la touche [Ctrl] enfoncée. Le maintien sur la touche [Ctrl] est indiqué par le “-” qui suit le “C”. “M-” désigne la touche “Meta” On l'utilise de la même manière que la touche [Ctrl].
Pour les claviers qui n'ont pas de touche [Meta], la touche [Alt] (ou la touche marquée d'un “◊” sur les claviers des machines Sun) joue souvent le même rôle. Sinon, on peut simuler M-x en tapant sur la touche [Esc] avant de taper sur la touche x. Les commandes

indiquées par C-M- dans le manuel s'obtiennent alors en tapant d'abord [Esc] puis [Ctrl] et le caractère suivant (et non pas l'inverse).

Commandes emacs

Toutes les actions sous emacs sont exécutées par des commandes dont le nom rappelle le type de l'action exécutée :
next-line
load-file
kill-word
goto-line
save-buffers-kill-emacs
L'utilisateur peut lancer ces commandes par leur nom en les préfixant avec M-x. Par exemple,
M-x kill-word

Associations de clés

Les actions les plus couramment employées sont associées à des clés (combinaisons de touches du clavier). Il suffit de taper la clé pour lancer la commande associée. Voici deux exemples :

Commande	Clé
next-line	C-n
save-buffers-kill-emacs	C-x C-c

Les autres commandes comme “goto-line” doivent être lancées avec M-x.
Les associations (“bindings”) peuvent être modifiées (voir 12.4) et peuvent dépendre du mode de travail (voir 12.2.5).

Compléter automatiquement des noms de commandes

Certaines commandes ont des noms assez longs (par exemple “goto-line”). Il existe des facilités pour ne pas être obligé de taper les noms en entier : on tape [Tab] (touche de tabulation) dès que le nom de la commande peut se distinguer des autres noms de commandes ; si on tape [ESP] (barre d'espace), emacs complète un mot dès que ce mot se distingue des autres mots clés. Pour l'exemple de “goto-line”, il suffit de taper “g” suivi de [ESP] pour avoir goto- et “l” suivi de [TAB]. Pour entrer une tabulation ou un espace, il suffit de les faire précéder de C-q.
Si emacs ne peut compléter parce qu'il y a plusieurs possibilités de complétions, il affiche une fenêtre d'aide où sont affichées toutes les suites possibles. On peut alors parcourir cette fenêtre par C-M-v. On peut choisir une des commandes en cliquant sur son nom avec

bouton du milieu de la souris. Cette fenêtre se referme automatiquement quand on lance la commande complète.

De même, les noms de fichiers tapés dans le mini-buffer (par exemple pour donner le nom d'un fichier à charger dans un buffer) peuvent être complétés par emacs quand l'utilisateur tape la touche [Tab] ou la barre d'espace.

12.2.3 Buffers et fenêtres

Tampon (ou buffer)

Chaque texte édité est conservé en mémoire centrale dans un buffer. Plus généralement, toute activité de emacs (aide en ligne, courrier électronique, etc.) est effectuée dans un buffer.

On peut travailler avec plusieurs buffers en même temps. Chaque buffer a un nom. Le nom d'un buffer associé à un fichier est le nom terminal du fichier. Si on travaille avec plusieurs fichiers qui ont le même nom terminal, un numéro est accolé à la fin du nom du buffer.

Sous X Window, le menu "Buffers" permet de choisir un buffer.

Fenêtre emacs

emacs ouvre une première fenêtre au démarrage.

Par la suite l'utilisateur peut diviser cette fenêtre en plusieurs sous-fenêtres. Une fenêtre emacs visualise une portion de buffer et permet de travailler avec les données affichées. Deux fenêtres peuvent visualiser des portions différentes d'un même buffer ou visualiser des buffers différents.

Le menu "File" permet de gérer les fenêtres: "Split Window" pour découper la fenêtre en cours en deux, ou "One Window" pour fermer toutes les fenêtres sauf une (pour cela, on peut aussi, dans les versions graphiques cliquer avec le bouton du milieu de la souris sur la ligne d'information de la fenêtre choisie).

On peut accéder à une fenêtre en cliquant avec le bouton gauche dans la fenêtre.

Fenêtre X ("frame" dans le manuel en anglais)

Sous X Window, l'utilisateur peut ouvrir une nouvelle fenêtre X pour y afficher le buffer courant. Il peut ensuite y charger un autre buffer ou y visualiser une autre partie du buffer courant comme il le ferait pour une fenêtre emacs.

Chaque fenêtre X possède son propre menu.

Pour ouvrir une nouvelle fenêtre X, le plus simple est d'utiliser le menu "File", choix "Make New Frame". La fenêtre peut être fermée par le même menu, choix "Delete Frame".

12.2.4 Point d'insertion, marque, régions

Point d'insertion

Le point d'insertion indique l'endroit où les modifications apportées au texte par l'utilisateur seront effectuées (insertion de nouveaux caractères, suppression,...). Le point est situé entre deux caractères. Il est indiqué sur l'écran par un curseur affiché sur la position qui suit le point.

Chaque tampon a un point d'insertion.

Région

La région est la zone de texte comprise entre le point d'insertion et la marque.

La marque peut être positionnée dans le texte en tapant C-<barre d'espace>ou implicitement par des manipulations de la souris.

C-x h met tout le texte du document comme région
M-h met tout le paragraphe comme région

Utilisation de la région et de l'anneau de suppression

Plusieurs commandes permettent de manipuler une région (une zone de texte) et de l'enregistrer dans un "anneau de suppression".

emacs conserve dans un "anneau de suppression" (commun à toutes les fenêtres) les textes supprimés par les commandes qui suppriment plus d'un seul caractère (C-k, C-mais pas C-d, [DEL] ou [Backspace]). La commande M-w permet aussi d'ajouter des textes à cet anneau sans les supprimer :

C-w supprime la région (attention au w trompeur!) et la place dans l'anneau de suppression
M-w enregistre la région dans l'anneau de suppression
L'utilisateur peut ensuite réutiliser les textes ainsi conservés :
C-y colle le dernier texte placé dans l'anneau de suppression à l'endroit du point d'insertion
M-y remplace le texte précédemment collé par le texte placé juste avant dans l'anneau de suppression. On peut ainsi passer en revue tous les textes conservés jusqu'à trouver le texte que l'on veut.

Dans un environnement X Window, on peut utiliser le menu "Edit" pour supprimer, copier ou coller une région. On peut aussi utiliser la souris :

Double-clic sur le bouton gauche
enregistre le mot pointé par la souris dans l'anneau de suppression
Triple-clic sur le bouton gauche
enregistre la ligne pointée dans l'anneau de suppression
Bouton du milieu
colle à l'endroit du clic un texte précédemment copié ou coupé (idem C-y)

Bouton droit
copie le texte placé entre le point et l'endroit du clic pour être collé ensuite (idem M-w).
Si on clique 2 fois, le texte est coupé au lieu d'être copié (idem C-w).

“Glisser” (*drag* en anglais)
(c'est-à-dire, cliquer et se déplacer en maintenant un bouton de la souris enfoncé) avec le bouton gauche, du début à la fin d'un texte, permet de définir une région et de l'enregistrer dans l'anneau de suppression.

12.2.5 Mode de travail

Emacs permet de travailler en plusieurs modes de travail adaptés au type de texte que l'on tape et au type de travail auquel ce buffer est associé. Il existe par exemple un mode de travail pour écrire un programme en langage C et des modes pour envoyer et recevoir du courrier électronique depuis emacs.

Chaque fenêtre emacs a un mode de travail majeur en cours.

On peut aussi lancer des modes “mineurs” qui modifient légèrement le fonctionnement des modes majeurs.

Nous n'étudierons pas ces modes dans ce cours. Voici seulement une description rapide du mode “dired” qui peut être très utile.

On entre dans ce mode “directory edition” soit en “ouvrant” un fichier répertoire au lieu d'un fichier ordinaire, soit en appelant explicitement le mode par la commande “M-x dired-mode”.

Ce mode est très pratique pour se déplacer dans l'arborescence : on clique avec le bouton du milieu sur le nom du répertoire où l'on veut aller ou sur le fichier que l'on veut charger dans un buffer. Dans ce cas, l'affichage se fait dans une deuxième fenêtre. Si on tape “f” sur la ligne qui contient le curseur, l'affichage se fait dans la même fenêtre.

On peut aussi copier et supprimer des fichiers. On se reportera à l'aide en ligne d'emacs et aux menus offerts par ce mode pour en savoir plus.

12.3 Commandes

12.3.1 Commandes de base

Insertion

Par défaut emacs est en mode insertion : tout caractère non spécial (affichable) tapé est inséré à l'endroit du point.

C-q permet d'insérer un caractère non affichable tapé à la suite

Suppression

Backspace supprime le caractère avant le point

C-d supprime le caractère situé après le point (donc placé sous le curseur)

12.3. COMMANDES

C-k efface la fin de la ligne (située après le point) et la place dans l'anneau de suppression

Déplacer le point d'insertion

Les terminaux sont souvent configurés pour que l'utilisateur puisse utiliser les flèches pour se déplacer d'un caractère ou d'une ligne et les touches [Next] et [Prev] pour se déplacer d'un écran.

On peut aussi utiliser les commandes ci-dessous :

C-a va au début de la ligne

C-e va à la fin de la ligne

M-< va au début du fichier

M-> va à la fin du fichier

Dans un environnement X Window,

Bouton gauche

déplace le point à l'endroit du clic

On peut aussi utiliser la barre de défilement pour se déplacer.

Barre de défilement ou “ascenseur” (Scroll Bar)

Toutes les fenêtres X Window associées à emacs possèdent une barre de défilement verticale à droite de la fenêtre. Cette barre représente tout le texte. Un rectangle interne plus ou moins long, représente la portion du texte actuellement affichée.

On utilise la souris pour se déplacer dans le texte à l'aide de cette barre. Les utilisateurs de Windows sur PC ou du Macintosh remarqueront que la manipulation de la barre de défilement est différente sous emacs. Cette manipulation est identique à la manipulation des ascenseurs sous le client X Window *xterm* (voir cours sur X Window).

Bouton gauche

déplace en haut de la fenêtre la ligne en face de laquelle on a cliqué. Le texte est donc déplacé vers la fin du fichier d'au plus un écran. Plus on clique bas, plus on avance dans le texte.

Bouton du milieu

déplace à l'endroit du clic le rectangle interne à la barre de défilement. Le texte affiché est le texte correspondant à la position nouvelle du rectangle interne. Si on veut se déplacer vers la fin du fichier, on clique donc sous le rectangle interne.

On peut aussi cliquer sur le rectangle interne de l'ascenseur avec ce bouton et glisser pour se déplacer continuellement dans le texte.

Bouton droit

déplace à l'endroit du clic la ligne placée en haut de la fenêtre. Le texte est donc déplacé vers le début du fichier d'au plus un écran. Plus on clique bas, plus on remonte dans le texte.

Sauvegarder un fichier

Les fichiers chargés dans un buffer ne sont modifiés que lorsque l'utilisateur indique explicitement qu'il veut sauvegarder les modifications qu'il a effectuées sur le buffer. C-x C-s sauvegarde le buffer courant

Sous X Window le menu "File" permet aussi de sauvegarder les fichiers sous son nom actuel (*Save Buffer*) ou sous un autre nom (*Save Buffer As...*).

Remarque 12.1

L'ancienne version du fichier (s'il ne s'agit pas d'un nouveau texte) est conservée sous l'ancien nom suivi du caractère "~".

emacs effectue des sauvegardes automatiques à intervalles réguliers. Ces sauvegardes sont effectuées dans des fichiers dont le nom est le nom du fichier modifié entouré du caractère "#". Ces sauvegardes sont supprimées en cas de fin d'édition normale. S'il y a eu une sortie anormale d'emacs et si vous n'avez pu sauvegarder vos fichiers, ces sauvegardes automatiques vous permettent de récupérer les bonnes versions de vos fichiers. De plus, si vous voulez charger un des fichiers qui n'a pu être sauvegardé normalement et si emacs s'aperçoit qu'il y a un fichier "#" plus récent que le fichier que vous voulez charger, emacs vous envoie le message "Auto save file is newer; consider M-x recover-file". Vous pouvez alors taper "M-x recover-file" et récupérer automatiquement la version enregistrée dans la sauvegarde automatique.

Aide

Sous X Window, le menu "Help" permet d'accéder aisément à toutes les aides nécessaires.

Défaire ("undo")

Si l'utilisateur s'est trompé il peut annuler la plupart des commandes qu'il a tapé. Sous X Window, il peut utiliser pour cela le choix "Undo" du menu "Edit". Il utile de noter que C-g sort du minibuffer d'entrée des commandes en annulant la commande que l'utilisateur est en train de taper.

On peut ainsi se sortir d'un mauvais pas lorsqu'*emacs* vous affiche continuellement le même message dans le mini-buffer parce que vous avez lancé une commande par une mauvaise manœuvre. Il est alors souvent nécessaire de taper C-g, si nécessaire plusieurs fois, et en se plaçant avec la souris dans le minibuffer si *emacs* s'entête.

12.3.2 Autres commandes

Rechercher du texte

- C-s effectue une recherche incrémentale (les caractères sont recherchés au fur et mesure qu'ils sont tapés) du texte tapé vers la fin du buffer. Pour passer à l'occurrence suivante, on retape C-s. Pour arrêter la recherche, on tape la touche Return ou une commande de déplacement. On peut ensuite relancer la même recherche en tapant C-s C-s.
- C-r effectue une recherche incrémentale vers le début du fichier. Pour passer à l'occurrence suivante, on retape C-r.
- C-M-s effectue la recherche d'une expression régulière (voir chapitre 11).

Remplacer une chaîne de caractères par une autre

- M-% La question "Query replace" s'affiche. On doit alors entrer la chaîne de caractères à remplacer suivie de [Return]. Il s'affiche alors "with" et l'on entre la chaîne de remplacement suivie de [Return]. A chaque fois que la chaîne à remplacer est rencontrée, l'utilisateur peut taper :
 - y ou n pour remplacer ou non,
 - q pour arrêter de remplacer,
 - ! pour tout remplacer jusqu'à la fin,
 - C-h pour avoir de l'aide sur la réponse

12.4 Personnalisation de emacs

Le fichier "~/.emacs" est exécuté au démarrage de emacs. Il doit contenir du code Lisp. Les amateurs éclairés pourront consulter le manuel de programmation de emacs pour plus de précisions (environ 700 pages). On dispose aussi d'une description des variables et des fonctions Lisp disponibles grâce au menu "Help", "Describe Function" et "Describe Variable".

Certaines personnalisations ne sont pas difficiles à effectuer, même si l'on ne connaît pas le langage Lisp. Par exemple, les affectations de valeurs à des variables (on peut s'aider du menu "Help" "Describe Variable") ou les associations de touches sont très simples :

```
(setq inhibit-startup-message t)
affecte la valeur "t" (true) à la variable "inhibit-startup-message" pour éviter d'avoir le message initial de emacs

(global-set-key [find] 'isearch-forward)
associe la commande de emacs "isearch-forward" à la touche du clavier "Find" qui pourra ainsi être utilisée pour lancer une recherche. Les noms des touches reconnus par emacs sont affichés par la commande "C-h k" ou par le menu "Help" "Describe Key".
```

1. Plus exactement, *elisp*, une variante du langage Lisp

Si une erreur survient lors de l'exécution du fichier “.emacs” ou des différents modules appelés, on peut visualiser les messages d'erreurs en allant dans le buffer “*Messages*”. On peut aussi évaluer du code “elisp” d'un buffer en mode “Emacs-Lisp” en tapant “C-x C-e” avec le curseur placé juste derrière la parenthèse fermante du code “elisp”.

Un grand nombre de *packages* écrits en *elisp* par la communauté internationale des utilisateurs d'*emacs* permettent d'ajouter de nombreuses fonctionnalités aux possibilités de base offerte par la distribution officielle d'*emacs*. On peut trouver la plupart sur le site *ftp archive.cis.ohio-state.edu* dans le répertoire */pub/gnu/emacs/elisp-archive/packages*.

Si on travaille sous X Window, on peut aussi personnaliser quelques caractéristiques de son environnement (couleurs, présence d'un ascenseur sur les fenêtres X, etc.) avec les fichiers d'initialisation des ressources. Tapez “man emacs” pour plus de précisions².

Depuis la version 20 un environnement minimum peut être installé pour tous les utilisateurs d'*emacs*. Cet environnement est défini dans les fichiers *site-load.el* et *site-start.el* (cherchez-les dans l'arborescence où est installée la version 20 ; les commandes *locate* ou *find* sont faites pour cela). Cet environnement, s'il vous convient vous permet de réduire de façon significative la taille de votre fichier .emacs. Si des options ne vous conviennent pas, allez consulter ces fichiers pour voir comment changer les choix qui y ont été faits.

12.5 Compléments pour les versions graphiques

Copier ou coller entre fenêtres X Window

Quand un texte est copié ou coupé (dans l'anneau de suppression) il peut être collé dans une autre fenêtre X Window, même s'il ne s'agit pas d'une fenêtre associée à emacs. Inversement, on peut coller dans emacs un texte précédemment copié ou coupé dans une autre fenêtre X Window.

Choix de la police de caractères sous X Window

S-BD (Shift + bouton droit de la souris) affiche un menu qui permet de choisir la police utilisée. On choisira plutôt une police à espacement fixe du type “Courier 14” ou “7x14”.

². La plupart des versions d'emacs ne sont pas écrites au-dessus de la boîte à outils Xt et tous les fichiers de ressources ne sont pas utilisés

Chapitre 13

Manipulation des données des fichiers

13.1 Tri (sort)

```
sort [options] [fichiers...]
```

trie les lignes des fichiers (regroupées ensemble). La sortie se fait sur la sortie standard. Par défaut, les lignes viennent de l'entrée standard.

Les lignes sont triées selon la valeur d'une clé de tri formée d'une ou plusieurs zones extraites de chaque ligne. Par défaut le tri se fait sur les lignes entières selon l'ordre lexicographique (et selon le code ASCII) et selon l'ordre croissant par défaut.

Options pour indiquer la clé de tri :

+pos1 -pos2 permet d'indiquer le début et la fin d'une zone de la clé de tri. Le coupé “+pos1 -pos2” indique que la zone de tri commence au champ numéro pos1 + 1 et se termine au champ numéro pos2 + 1 (attention, non compris !). S'il n'y a pas de -pos2, la zone de tri va jusqu'à la fin de la ligne. La notation de champ est définie ci-dessous dans la définition de l'option -t.

-t séparateur indique le caractère qui sépare les différents champs.

Si un séparateur n'est pas défini, les champs sont séparés par toute suite d'espaces ou de tabulations contigus. Le premier caractère “blanc” est considéré comme un séparateur ; les espaces et tabulations suivants font partie du champ suivant (voir option -b).

Il peut y avoir plusieurs clés de tri (voir exemple (b) ci-dessous).

-b ignore les espaces de tête dans chaque champ (-b est placé avant tous les +pos1 ou b est accolé derrière les +pos1 des champs pour lesquels cette option est valable).

-n effectue un tri numérique (et non lexicographique) sur les clés. On peut aussi accoler la lettre n derrière +pos1 si on ne veut trier numériquement que certains champs.

-f ignore la différence entre majuscules et minuscules

Autres options :

-c affiche un message si les lignes ne sont pas dans le bon ordre (et dans ce cas

renvoie un code retour égal à 1 au lieu de 0)

-ofichier sort le résultat du tri dans *fichier*. Attention à l'erreur suivante :

```
sort fichier > fichier
```

qui détruit *fichier* avant de le trier (voir redirections en 14.2). La bonne commande est

```
sort -ofichier fichier
```

-r inverse l'ordre de tri. On peut aussi accoler la lettre r derrière +pos1 si on ne veut trier par ordre décroissant que certains champs.

-u si plusieurs lignes ont la même valeur pour la clé de tri, une seule de ces lignes est envoyée sur la sortie standard.

Exemples 13.1

- (a) Tri de `/etc/passwd` sur le numéro de l'utilisateur :

```
sort -t: +2n -3 /etc/passwd
```

Le tri ne se fait pas correctement si on oublie le n à la suite de +2.

- (b) Tri les fichiers du répertoire courant par taille et par nom pour les fichiers qui ont la même taille :

```
ls -lgA | sort +4n -5 +8
```

13.2 Recherche d'une chaîne de caractères (grep)

```
grep [options] expr-reg [fichiers...]
```

affiche sur l'écran les lignes des fichiers, qui contiennent une chaîne de caractères correspondant à l'expression régulière *expr-reg* (voir chapitre 11. L'entrée standard est prise par défaut si des fichiers ne sont pas spécifiés. Le code retour renvoyé est 0 si l'expression régulière a été trouvée ou 1 sinon.

Si *expr-reg* comporte des caractères spéciaux pour le shell (* ou [par exemple), il ne faut pas oublier de les protéger en les entourant d'apostrophes ou de guillemets (voir 14.8).

Options

-c affiche seulement le nombre de lignes contenant la chaîne

-i ignore la différence entre minuscules et majuscules

-n affiche les numéros des lignes

-v affiche les lignes qui ne contiennent pas la chaîne

Exemple 13.2

```
grep -c '~bin' fichier
```

affiche le nombre de lignes de fichier qui commencent par "bin".

13.3 Compter les caractères, les mots, les lignes (wc)

```
wc [options...] [fichiers...]
```

13.4. CONVERSION, SUPPRESSION DE CARACTÈRES (tr)

affiche le nombre de lignes, de mots, de caractères et le nom de chaque fichier. Si des fichiers ne sont pas spécifiés, l'entrée standard est prise par défaut.

Options

-l nombre de lignes seulement

-w nombre de mots seulement

-c nombre de caractères seulement

Exemples 13.3

- (a) Nombre de fichiers de /bin :

```
ls -A /bin | wc -l
```

- (b) Compter les lignes, mots et caractères de tous les fichiers du répertoire courant dont le nom commence par b :

```
wc b*
```

13.4 Conversion, suppression de caractères (tr)

La commande "tr" permet de convertir ou de supprimer des caractères provenant de l'entrée standard.

Conversion de caractères

```
tr [-cs] chaîne1 chaîne2
```

transforme les caractères provenant de l'entrée standard ainsi : les caractères contenus dans *chaîne1* sont transformés en les caractères correspondants de *chaîne2* et les autres caractères sont conservés tels quels. Le tout est envoyé vers la sortie standard.

Si *chaîne2* contient moins de caractères que *chaîne1*, le dernier caractère de *chaîne2* remplace les caractères manquants.

On peut désigner un caractère par son code ASCII en octal précédé par \. On peut aussi désigner un intervalle de caractères : **e-l** désigne les lettres minuscules comprises entre e et l (bornes comprises). En Unix système V, la syntaxe est plutôt **[e-l]**.

Exemples 13.4

(">" est le prompt dans les exemples suivants)

```
~> tr aeiou AEIOU
bonjour
[Ctrl] D
bOnjOUr
```

- (b) ~> tr "a-z" "A-Z"

```
abcd
[Ctrl] D
ABCD
```

(c) `cat toto | tr ae AE > totobis`
ou
`tr ae AE > totobis < toto`

Options

- c prend pour *chaîne1* le complément de la *chaîne1* indiquée dans la commande. Le complément est pris par rapport à l'ensemble des caractères de code 0 à 255.
- s (squeeze) ne garde en sortie qu'un seul exemplaire d'une suite d'un même caractère de *chaîne2*.

Exemple 13.5

```
tr -cs a-zA-Z '\012'
```

affiche l'entrée au clavier, un “mot” (chaîne composée uniquement de lettres) par ligne (\012 est le caractère “passage à la ligne” de code 10 en décimal, 012 en octal).

Suppression de caractères

```
tr -d chaîne
```

envoie vers la sortie standard les caractères reçus dans l'entrée standard s'ils ne sont pas inclus dans chaîne. *tr* supprime au passage tous les caractères inclus dans chaîne.

Exemple 13.6

```
~> tr -d ac
abcd
[Ctrl] D
bd
~>
```

13.5 Fractionnement vertical (cut)

```
cut -cliste [fichiers...]
```

découpe un ou plusieurs fichiers (entrée standard par défaut) verticalement en sélectionnant des intervalles de *caractères* (comme l'indique l'option “-c”) à conserver dans chaque ligne. Le résultat est envoyé sur la sortie standard.

liste est une liste de nombres entiers séparés par des virgules avec éventuellement des tirets pour indiquer un intervalle. Le découpage se fait sur les numéros de caractères (le premier caractère a le numéro 1). Cette liste indique quels seront les caractères qui seront conservés sur chaque ligne.

Exemples 13.7

- (a) 1,3,8
désigne les caractères situés en position 1, 3 et 8
- (b) 2-5,10
désigne les caractères situés en position 2 à 5 et 10

13.5. FRACTIONNEMENT VERTICAL (CUT)

- (c) -5,8
désigne les caractères situés en position 1 à 5 et 8
- (d) 3-
désigne les caractères situés en position 3 à la fin de la ligne
- (e) 12-20,4-15
désigne les caractères situés en position 4 à 20

Les caractères seront toujours affichés dans l'ordre qu'ils ont sur la ligne du fichier, quel que soit l'ordre donné dans la liste (voir exemple b ci-dessous).

Exemples 13.8

- (a) `cut -c-5,10-15 fichier1 fichier2`
extrait les 5 premiers caractères et les caractères 10 à 15 de chaque ligne de *fichier1* et de *fichier2* et les affiche sur l'écran
- (b) `cut -c10-12,4-11 fichier`
extrait les caractères 4 à 12 de chaque ligne. On remarquera que les caractères 10 et 11 ne sont affichés qu'une seule fois.

```
cut -fliste [-ddélimiteur] [-s] [fichiers...]
```

dans ce cas le découpage se fait sur des champs (comme l'indique l'option “-f”) délimités par *délimiteur* (option “-d” ; tabulation par défaut).

Attention, si plusieurs délimiteurs sont accolés, ils définissent plusieurs champs. C'est une source fréquente d'erreurs quand le délimiteur est la tabulation ou l'espace. Quand on veut extraire des champs séparés par un ou plusieurs espaces il est préférable d'utiliser la commande *awk* (voir 13.10).

Les lignes sans délimiteur sont conservées en entier. L'option -s indique qu'il ne faut pas conserver les lignes sans délimiteur.

liste est une liste de nombres entiers séparés par des virgules avec éventuellement des tirets pour indiquer un intervalle. Le découpage se fait sur les numéros de champs (le premier champ a le numéro 1). Cette liste indique quels seront les champs qui seront conservés sur chaque ligne.

Les caractères seront toujours affichés dans l'ordre qu'ils ont sur la ligne du fichier, quel que soit l'ordre donné dans la liste.

Exemples 13.9

- (a) `cut -f1,5 -d: /etc/passwd`
affiche les utilisateurs et le commentaire associé du fichier */etc/passwd*
- (b) Compter combien il y a de clients distincts dans un fichier de factures dont chaque ligne correspond à un client placé en tête de ligne (tabulation entre chaque zone d'une ligne):

```
cut -f1 factures | sort -u | wc -l
```

13.6 Comparaison du contenu de 2 fichiers

13.6.1 Différences entre deux fichiers texte (diff)

```
diff [-e] fichier1 fichier2
```

indique quelles lignes doivent être changées dans *fichier1* pour qu'il soit identique à *fichier2*. Les deux fichiers doivent être de type texte, structurés en lignes (voir 1.6.2).

On peut ainsi comparer deux versions d'un même fichier.

Le résultat n'est pas toujours facile à comprendre. La lettre "a" indique que des lignes ont été ajoutées à *fichier2* par rapport à *fichier1*, "d" indique que des lignes ont été supprimées, "c" que des lignes ont été changées. Les nombres avant la lettre sont des numéros de lignes de *fichier1* et les numéros qui suivent la lettre sont des numéros de lignes de *fichier2*. Les lignes de *fichier1* sont précédées de "<", celles de *fichier2* sont précédées de ">".

Exemple 13.10

```
~> cat f1
ab
c
e
fg
h
~> cat f2
ab
e
f
u
~> diff f1 f2
2d1
< c
4,5c3,4
< fg
< h
---
> f
> u
```

13.6.2 Égalité du contenu de 2 fichiers (cmp)

```
cmp [-s] fichier1 fichier2
```

affiche le premier octet différent dans les 2 fichiers.

Option

-s cmp n'affiche rien mais retourne seulement un code de retour (voir 14.1.6) : 0 si les 2 fichiers sont identiques, 1 s'ils sont différents et 2 s'il y a eu une erreur.

13.7 TRAITER LES LIGNES CONSÉCUTIVES IDENTIQUES (UNIQ)

Exemple 13.11

```
$ cmp fich1 fich2
fich1 fich2 differ : char 160, line 5
```

13.7 Traiter les lignes consécutives identiques d'un fichier (uniq)

```
uniq [-cdu] [fichier]
```

n'affiche qu'un seul exemplaire des lignes de [*fichier*] (entrée standard par défaut). Si plusieurs lignes *consécutives* sont identiques, une seule est envoyée vers la sortie standard. Cette commande ne considère pas deux lignes identiques si elles ne sont pas consécutives (il faut faire un tri préalable si on veut les prendre en compte).

Option

-c affiche chaque ligne en un seul exemplaire, en la faisant précéder du nombre de lignes voisines identiques
-d n'affiche que les lignes identiques (en un seul exemplaire)
-u n'affiche que les lignes en un seul exemplaire

13.8 Extraire le début ou la fin d'un fichier

13.8.1 Début d'un fichier (head)

```
head [-n] [fichiers...]
```

sort sur la sortie standard les *n* premières lignes de chacun des *fichiers* (de l'entrée standard par défaut). Par défaut, *n* est égal à 10.

Exemple 13.12

```
head -5 fich*
```

13.8.2 Fin d'un fichier (tail)

```
tail [début] [fichier]
```

copie la fin du fichier sur la sortie standard.

La première ligne affichée est indiquée par *début* qui peut être $+n$ ($n^{\text{ème}}$ ligne du fichier) ou $-n$ ($n^{\text{ème}}$ ligne à partir de la fin du fichier).

n peut aussi être suivi des lettres b ou c qui indiquent que l'on compte en blocs ou en caractères et non pas en lignes.

Si le fichier n'est pas indiqué, tail prend l'entrée standard par défaut.

Exemples 13.13

(a) **tail +120 fich**

(b) **tail -2b fich**

13.9 Un éditeur non interactif (sed)

13.9.1 Description générale de l'éditeur

sed est un éditeur non interactif de fichier : il lit les données provenant d'un fichier (ou de l'entrée standard par défaut), les traite suivant un programme et sort le résultat du traitement sur la sortie standard.

sed est souvent utilisé comme filtre dans un pipe.

Comme pour la commande *awk* vue à la section suivante, nous ne verrons qu'une petite partie des possibilités offertes par cette commande.

```
sed [-n] -e commande [fichiers...]
```

copie les *fichiers* (l'entrée standard par défaut) sur la sortie standard après avoir effectué le traitement indiqué dans *commande*. *commande* doit être un mot pour le shell (donc entouré de guillemets ou d'apostrophes s'il contient des espaces). Il peut y avoir plusieurs options “-e”. S'il n'y en a qu'une, “-e” est optionnel.

```
sed [-n] -f fichier-pgm [fichiers...]
```

indique que le programme est enregistré dans un fichier.

L'option -n indique que les lignes lues ne doivent pas être automatiquement renvoyées sur la sortie standard après leur traitement. Seules les lignes explicitement envoyées par la commande p ou une commande similaire le seront (il faut par exemple ajouter l'option “p” de la commande “s”).

Une grande partie des traitements effectués par sed peut être exécutée par awk. awk est plus puissant mais sed est plus simple d'emploi pour remplacer une chaîne de caractères par une autre (commande “s”).

13.9.2 Structure d'une ligne de programme

```
lignes commande arguments
```

“lignes” indique les lignes sur lesquelles la commande *sed* sera appliquée. L'argument permet de préciser le traitement à effectuer sur les lignes.

13.9.3 Format pour indiquer les lignes à traiter

Les lignes sont numérotées de 1 au nombre de lignes reçues.

L'adresse d'une ligne peut être désignée par un numéro de ligne, par \$ (dernière ligne) ou par une expression régulière entourée de /.

Les lignes traitées peuvent être désignées par :

adresse1, adresse2

qui désigne un ou plusieurs blocs de lignes comprises entre les lignes désignées par les deux adresses

adresse

qui désigne toutes les lignes correspondant à l'adresse

13.10. UN LANGAGE D'ÉDITION DE FICHIERS (AWK)

aucune adresse

qui désigne *toutes* les lignes.

13.9.4 Exécution du programme

sed applique chaque commande du programme à la première ligne du fichier traité. Elle passe ensuite à la ligne suivante et ainsi de suite.

Chaque commande s'applique à la version de la ligne courante déjà modifiée par les commandes déjà exécutées sur cette ligne.

13.9.5 Commandes

Les commandes les plus utilisées sont

s pour substituer une chaîne de caractères à une autre,

d pour supprimer des lignes,

p pour envoyer des lignes vers la sortie standard.

On peut regrouper plusieurs commandes avec { et }.

Exemple 13.14

```
for dir in `echo $PATH | sed -e 's:/ /g'`
do .....
```

Une grande partie des traitements effectués par *sed* peut être exécuté par *awk*. *awk* est plus puissant mais *sed* est plus simple d'emploi pour certaines tâches, en particulier pour remplacer une chaîne de caractères par une autre (commande “s”).

On remarquera que le fichier traité n'est pas modifié par *sed*. Si on veut modifier un fichier de façon non interactive, le plus simple est d'utiliser l'éditeur *ed* avec un fichier inclus (voir 16.4.1 ; la commande “s” y est détaillée).

13.10 Un langage d'édition de fichiers (awk)

13.10.1 Description générale de la commande

awk¹ lit un texte contenu dans un ou plusieurs fichiers, le modifie et envoie le résultat vers la sortie standard.

awk est une commande très puissante pour travailler sur le contenu d'un fichier. Elle possède un langage de programmation comportant des structures conditionnelles, des boucles et des fonctions prédéfinies ; ce langage peut travailler avec des variables et des tableaux. Nous n'étudierons ici que les fonctionnalités les plus simples offertes par awk.

1. du nom de ses développeurs Aho, Weinberger et Kernighan

Il est fortement conseillé d'étudier plus en détail cette commande si l'on veut effectuer des traitements complexes sur le contenu des fichiers. Un langage encore plus puissant est fourni par le langage *Perl* qui ne sera pas étudié ici.

Le traitement à effectuer est écrit dans un programme tapé directement dans la ligne de commande (forme 1 de la syntaxe) ou enregistré dans un fichier (forme 2 de la syntaxe).

```
awk [-Fc] programme [fichiers...]
ou
awk [-Fc] -f fichier-programme [fichiers...]
```

L'option -F permet d'indiquer le séparateur de champ (voir 13.10.2).

Pour la première forme de la syntaxe, le programme doit être considéré comme un seul “mot” par le shell ; en général, il est donc nécessaire de l'entourer par des apostrophes ou des guillemets.

Exécution de la commande :

awk travaille sur des enregistrements constitués de champs. Le séparateur d'enregistrement étant par défaut le séparateur de ligne de Unix, awk travaille ordinairement sur des lignes.

awk lit enregistrement par enregistrement les données des fichiers (l'entrée standard par défaut). Chaque enregistrement est traité à son tour selon les indications données par toutes les instructions du programme, avant la lecture de l'enregistrement suivant.

13.10.2 Champs

Quand awk lit un enregistrement d'un fichier, il l'éclate en champs séparés par des espaces ou des tabulations (par défaut) ou par le caractère suivant l'option -F.

Le programme peut faire référence à un ou plusieurs champs de la ligne lue :

```
$1, $2, ...
```

représentent les 1^{er}, 2^{ème}, ... champs

```
$0
```

représente la ligne entière.

Exemples 13.15

- (a) Afficher le nom, les autorisations et le propriétaire des sous-répertoires du répertoire courant :

```
ls -lg | awk '/^d/ { print $9, $1, $3 }'
```

- (b) Afficher les noms des utilisateurs :

```
awk -F: '{print $1}' /etc/passwd
```

13.10.3 Structure d'un programme

Un programme peut être complexe et comporter de nombreuses lignes incluant toutes les structures de programmation que l'on trouve habituellement dans les langages de programmation comme le langage Pascal ou le langage C.

13.10. UN LANGAGE D'ÉDITION DE FICHIERS (AWK)

Une ligne d'un programme a la structure suivante :

```
sélecteur { action }
```

sélecteur indique si *action* doit être exécutée sur l'enregistrement en cours de traitement.

Dans ce cours nous ne verrons que les programmes comportant une seule ligne qui affiche (par la commande print) des portions de lignes du fichier traité.

Les constantes de type chaîne de caractères sont entourées par des guillemets.

On peut inclure des *commentaires* dans une ligne du programme en les faisant précéder du caractère “#”.

13.10.4 Cas particuliers pour une ligne de programme

- S'il n'y a pas de *sélecteur*, *action* est toujours exécutée (pour tous les enregistrements).

Exemple 13.16

```
ls -lg | awk ' { print $9, $5 }'
```

affiche le nom et la taille (les neuvième et cinquième champs de la ligne) de tous les fichiers du répertoire courant.

- S'il n'y a pas d'*action*, l'enregistrement traité est recopié vers la sortie s'il a été sélectionné par *sélecteur*.

Exemple 13.17

L'exemple de 13.18 pourrait donc s'écrire plus simplement.

```
ls -l | awk '/^d/'
```

13.10.5 Sélecteurs

Ils peuvent être du type :

- /expression régulière/

Dans ce cas, *action* sera exécutée sur un enregistrement s'il contient une chaîne de caractères qui correspond à *expression-régulière* (voir 11).

Exemple 13.18

```
ls -l | awk '/^d/ { print }'
```

affiche toutes les lignes de “ls -l” concernant les sous-répertoires du répertoire courant.

- expression de comparaison (semblables à celles du Langage C) :

```
==, !=, <, >, <=, >=
```

De plus, ~ (tilde) permet de comparer une chaîne de caractères à une expression régulière (placée entre deux “/”). Par exemple,

```
$5 ~ /^[0-9]/
```

- expression composée des deux types précédents reliés par des opérateurs logiques
|| (ou), && (et), ! (négation de l'opérateur qui suit)
- mots clé BEGIN ou END. Si le sélecteur de la première (resp. dernière) ligne du programme est BEGIN (resp. END), l'action de cette ligne sera exécutée une fois avant (resp. après) de traiter le premier (resp. dernier) enregistrement des fichiers. Des exemples d'utilisation de ces mots clés sont donnés en 13.11.

Exemples 13.19

- (a) `$2 == ""`
- (b) `$2 == "1" || $2 == "5"`
- (c) `$2 !~ /^ab/` `# deuxieme champ ne commence pas par ab`
- (d) `$2 > 3`
- (e) `/^$/` `# ligne vide`

Le format général d'un programme est donc :

```
BEGIN { instruction de début }
sélecteur { action }
....
....
sélecteur { action }
END { instruction de fin }
```

13.10.6 Actions

Une action est une suite d'instructions. Comme en Langage C plusieurs actions peuvent être regroupées entre accolades. Une instruction se termine par un “;”, un passage à la ligne ou par une accolade fermante.

Une instruction peut être une instruction de contrôle, une affectation, print ou printf.

Les instructions de contrôle disponibles sont :

if, while, for, break, continue, next, exit.

L'instruction *next* fait passer directement au traitement de la ligne suivante.

Pour les autres instructions, la syntaxe est semblable à celle du langage C. Les instructions composées de plusieurs instructions sont entourées d'accolades :

```
{ if (n > 1000) {
    i++
    somme += $4
  }
}
```

13.10. UN LANGAGE D'ÉDITION DE FICHIERS (AWK)

Pour une description plus détaillée, on se reportera au manuel de référence ou, encore mieux, à un guide d'utilisation de *awk*.

Des exemples d'utilisation des instructions de contrôle sont donnés dans la section suivante.

Voyons un peu plus en détails print et printf :

```
print [expressions...] [>fichier]
ou
print [expressions...] [>>fichier]
```

affiche les expressions sur la sortie standard ou vers le fichier éventuellement spécifié et redirection. print effectue ensuite un passage à la ligne.

Si les expressions qui suivent “print” sont séparées par un ou plusieurs espaces, elles sont sorties accolées les unes derrière les autres. Si elles sont séparées par une virgule, elles sont sorties séparées par un espace.

On peut ajouter un pipe en sortie de print. La commande Unix qui reçoit ce qui est envoyé par print doit être entourée de guillemets.

“print” (seul) affiche l'enregistrement complet.

Variables utilisateur

Des variables peuvent être utilisées dans les programmes. La syntaxe de leur utilisation est semblable au Langage C. Les variables sont initialisées à la chaîne vide. Les variables qui sont utilisées dans des calculs sont automatiquement initialisées à 0.

Exemple 13.20

Afficher la taille totale des fichiers affichés par la commande ls :

```
ls -lg |
awk ' { print $9, $5
      somme = somme + $5 }
    END { print "Total des tailles des fichiers :", somme }'
```

Remarque 13.1

Les constantes chaînes de caractères doivent être entourées par des guillemets simples, elles sont considérées comme des noms de variables par *awk*. C'est une source d'erreurs fréquente lorsque l'on commence à programmer avec *awk*.

Variables système

Voici quelques variables système utilisées par *awk* :

NF	nombre de champs de la ligne
NR	nombre de lignes qui ont été lues
FS	séparateur de champs en entrée (par défaut l'espace ou la tabulation)
OFS	séparateur de champs en sortie (par défaut l'espace)

RS séparateur d'enregistrements (line feed de code 10 par défaut).
 “**RS=""**” indique que le séparateur d'enregistrement est la ligne vide. C'est utile pour travailler avec des enregistrements multi-lignes.

FILENAME nom du fichier en cours de traitement
 FS, OFS et RS sont modifiables par le programme.

13.10.7 Fonctions, opérateurs

On peut utiliser des fonctions (de syntaxes semblables aux fonctions du Langage C) dans les sélecteurs et les actions.

Pour faire des calculs :

int, sqrt, log, exp, int,...

Pour travailler avec des chaînes de caractères :

substr, length, index,...

La fonction getline permet de lire une ligne ou la valeur d'une variable :

“getline” lit une ligne du fichier en cours.

“getline *variable*” lit une ligne du fichier en cours et met la valeur dans *variable*.

On peut spécifier en entrée un autre fichier pour “getline” par une redirection de l'entrée :

“getline [*variable*] < *fichier*”

La valeur lue par getline peut aussi provenir d'une commande Unix :

“*commande* | getline [*variable*]”

Pour lancer une commande Unix, on dispose de la fonction “*system*”. Par exemple,

```
system("rm fichier")
```

On peut utiliser aussi des calculs avec les opérateurs du langage C :

+, -, *, /, %, ++, --, +=, -=, /= et %=

Exemple 13.21

```
length($0) > 72 { print "Ligne", NR, "longue:", substr($0, 1, 72) }
```

Les versions *nawk* et *gawk* ont les fonctions gsub et sub qui permettent de remplacer une chaîne de caractères par une autre dans les lignes du fichier.

13.11 Exemples de programmes

– Afficher le nombre de lignes du fichier :

```
awk 'END { print NR }' fichier
```

– Afficher le premier et le dernier champ de chaque ligne :

```
awk '{ print $1, $NF }' fichier
```

– Afficher le nom et la taille des fichiers du répertoire courant :

```
ls -alg | awk '{ print $9, $3 }'
```

13.11. EXEMPLES DE PROGRAMMES

– Afficher le nom et la taille des fichiers du répertoire courant avec un format de sortie plus agréable (le nom cadré à gauche dans un champ de 20 caractères et la taille cadrée à droite dans un champ de 6 caractères) :

```
ls -alg | awk '{ printf "%-20s %6d\n", $9, $4 }'
```

– Afficher les lignes qui contiennent ab ou cde :

```
awk '/ab/ || /cde/'
```

– Afficher tous les intervalles de lignes qui sont incluses entre deux lignes égales “debut” et à “fin” :

```
awk '/~debut$/~/~fin$/'
```

– Afficher les lignes dont le premier champ est différent de celui de la ligne précédente :

```
awk '$1 != prev { print ; prev = $1 }'
```

– Voici un fichier programme “adjacents” que l'on peut appeler par

```
awk -f adjacents fichier
```

pour repérer les mots adjacents identiques dans un fichier :

```
NF>0 {
  if ($1 == derniermot)
    print "double %s, ligne %s\n", $1, NR
  for (i=2 ; i<=NF ; i++)
    if ($i == $(i-1))
      printf "double %s, ligne %s\n", $i, NR
  if (NF>0)
    derniermot = $NF
}
```

– Afficher les groupes auxquels l'utilisateur appartient :

```
awk -F: /[:,] 'logname' ($1,)/ { printf "%s\n", $1 }' /etc/group
```

Remarquez l'expression “(\$1,)” qui indique que le nom d'utilisateur est suivi par une virgule ou par la fin de la ligne.

– “Plier” les lignes qui ont plus de 50 caractères dans un fichier (on ajoute un \ à droite des lignes pliées :

```
length($0) > 72 {
  print substr($0,1,50) "\\ "
  ligne = substr($0,51)
  while (length(ligne) > 50) {
    print substr(ligne,1,50) "\\ "
```



```

    ligne = substr(ligne,51)
  }
  print ligne
}
length($0) < 72

```

13.12 Tableaux

awk peut manier des tableaux dont les indices ne sont pas nécessairement des entiers.

Voici un exemple de programme utilisant un tableau pour additionner toutes les valeurs correspondant à des noms d'un fichier de la forme :

```

toto  100
bibib  200
toto  150
jean  200
toto  300
bibib  150

```

Le programme affichera :

```

toto  550
bibib  350
jean  200

```

Voici le programme :

```

{ somme[$1] += $2 }
END { for (nom in somme) print nom, somme[nom] }

```

Remarquez le “`for (nom in somme)`” qui parcourt le tableau.

On peut utiliser l'élément d'un tableau comme condition d'un if. Dans ce cas, la condition est vraie si l'indice correspond à un élément qui a reçu une valeur dans le programme :

```

{ if (valeur[$1]) print $1, "a déjà une valeur" }

```

La fonction *split* permet de découper une chaîne de caractères en affectant chaque champ de la chaîne aux éléments d'un tableau.

13.13 D'autres commandes de manipulation de fichiers

split	Fractionnement horizontal d'un fichier en fichiers de tailles fixées (on peut recoller les morceaux par <code>cat</code>)
csplit	Fractionnement horizontal d'un fichier selon le contenu des lignes du fichier (on peut recoller les morceaux par <code>cat</code>)
fold	découpe les lignes trop longues en plusieurs lignes
paste	Recollement vertical (en colonnes) de plusieurs fichiers (le contraire de <code>cut</code>)
comm	Sélection/rejet de lignes communes à 2 fichiers
join	Jointure sur une zone commune (au sens des bases de données relationnelles) de 2 fichiers triés suivant cette zone

Troisième partie

Mécanismes d'interprétation du shell

Chapitre 14

Interprétation du shell. Mécanismes de base

14.1 Généralités sur le shell

14.1.1 Définition

Quand l'utilisateur tape son nom et son mot de passe, le programme de démarrage associé à l'utilisateur (voir 2.1.1) est automatiquement lancé. Ce programme est le plus souvent un shell et toute la session de l'utilisateur se déroule en fait dans ce shell.

Quand une commande est tapée au clavier, le programme “shell” la lit, l'interprète et lance son exécution. La ligne tapée au clavier peut ne comporter qu'un nom de commande mais elle peut aussi être beaucoup plus complexe car le shell comprend un véritable langage de programmation qui apporte beaucoup de souplesse et de puissance à l'utilisateur.

L'utilisateur peut d'ailleurs écrire des programmes dans le langage du shell et les enregistrer dans des fichiers appelés *shellscripts* (voir 14.11).

14.1.2 Compatibilité des différents shells

Nous travaillerons avec le shell *zsh*. Sous ce shell les shellscripts sont exécutés par le shell *sh* sauf mention contraire explicite du programmeur. Le shell standard est actuellement *ksh*.

Nous étudierons donc essentiellement les commandes communes à *sh*, à *ksh* et à *zsh*, en indiquant les cas où les shells ne fonctionnent pas exactement de la même façon.

Autant que possible on essaiera de n'utiliser que les commandes et possibilités offertes par tous les shells quand on écrira un shellscript, pour favoriser sa portabilité. En interagissant avec le shell, le problème ne se pose pas et on pourra utiliser toutes les facilités du shell avec lequel on travaille.

Les shells peuvent être configurés de différentes façons selon les valeurs d'options qu'on leur donne. On peut donner, soit à l'appel du shell, soit par la commande *set* (le plus souvent lancée dans un fichier d'initialisation du shell ; voir 15.3). Par exemple,

```
zsh -o NOCLOBBER
ou
set -o NOCLOBBER
```

positionne l'option NOCLOBBER (voir 14.2.1) au démarrage ou durant une session de *zsh*.

Pour avoir une bonne compatibilité de *zsh* avec *ksh* et *sh*, il faut positionner l'option NO_NOMATCH (voir 4.5) et ne pas positionner les options GLOBDOTS et NOCLOBBER. Dans ce cas, l'incompatibilité principale de *zsh* avec les autres shells provient de la façon d'indiquer la “négation” de caractères pour la génération des noms de fichiers : `[!]` pour *sh* et *ksh* et `[^]` pour *zsh*.

14.1.3 Commandes et processus, commandes internes au shell

Suivant les cas, une ligne tapée au clavier peut lancer aucun, un ou plusieurs nouveaux processus (voir 2.2) :

- La plupart des lignes de commandes tapées par l'utilisateur entraînent l'exécution d'un fichier binaire enregistré dans l'arborescence des fichiers. Le shell lance l'exécution de ce fichier après avoir interprété la ligne de commande. Cette exécution engendre un nouveau processus.
- Certaines lignes de commandes peuvent engendrer plusieurs processus (par exemple les lignes de commandes comportant un pipe).
- D'autres commandes sont internes au shell ; le shell sait les exécuter sans faire appel à un fichier externe et aucun nouveau processus n'est engendré. Ces commandes sont peu nombreuses : essentiellement *cd* et *pwd* dans les commandes les plus courantes.

14.1.4 Prompts

Le shell indique qu'il est prêt à prendre une nouvelle commande en affichant le *prompt* (par défaut le prompt est “\$”).

Une commande se termine par un passage à la ligne. Dans certains cas le passage à la ligne n'est pas considéré comme un terminateur de commande par le shell ; dans ces cas, il s'affiche un deuxième prompt (“> ” par défaut) pour indiquer que le shell attend la suite de la commande :

```
$ echo "LIGNE 1
> LIGNE 2"
LIGNE 1
LIGNE 2
```

14.1.5 Environnement d'une commande ou d'un shellscript

Toutes les commandes lancées dans Unix s'exécutent dans un environnement de travail défini par un ensemble de variables (voir chapitre 15).

Cet environnement peut guider les commandes pendant leur exécution. Par exemple, la commande *lpr* envoie les requêtes d'impression vers l'imprimante dont le nom est contenu dans la variable `PRINTER` si cette variable fait partie de l'environnement.

14.1.6 Code retour d'une commande

Toute commande Unix renvoie un code retour.

En général, le code retour est égal à 0 si tout s'est bien passé et à un entier strictement positif sinon. Il permet d'automatiser des tâches à l'aide de *shellscripts* en tenant compte des résultats des commandes comme nous le verrons dans le chapitre 17.

Remarque 14.1

Ce code retour ne s'affiche pas¹ ; il est enregistré dans la variable “?” du shell (voir 15.2.6). Il peut être affiché par la commande “`echo $?`”.

La touche d'annulation (souvent `[Ctrl] C` ; voir 3.3) permet d'interrompre l'exécution d'une commande si elle n'a pas été lancée en arrière-plan. Le code retour est alors en général différent de 0 (cela dépend de la manière dont a été écrite la commande ; voir commande interne *trap* en 17.5).

14.2 Redirections

Avant de lancer un processus, le shell² lui associe 3 fichiers :

Entrée standard (par défaut le clavier) ; le descripteur de fichier est 0.

Sortie standard (par défaut l'écran) ; le descripteur de fichier est 1.

Erreur standard (par défaut l'écran) ; le descripteur de fichier est 2.

Ces 3 descripteurs peuvent être liés à d'autres fichiers que ceux indiqués ci-dessus grâce aux commandes de *redirection*. Le shell va associer d'autres fichiers que le clavier ou l'écran à ces 3 descripteurs et il lancera ensuite la commande.

14.2.1 Redirections de la sortie standard

Redirection avec écrasement du fichier de redirection (>)

On redirige la sortie standard d'une commande dans un autre fichier que le fichier spécial associé à l'écran, par la redirection
`>fichier`

1. Sous *zsh*, l'option “`print_exit_value`” fait afficher le code retour s'il est différent de 0

2. C'est une convention adoptée par tous les shells et par l'immense majorité des programmes écrits pour Unix

Cette redirection associe le fichier *fichier* au descripteur 1. Tout ce qui est envoyé vers la sortie standard (connue par la commande `comme` le fichier de descripteur 1) est donc envoyé à *fichier*. La redirection doit être écrite à la suite de la commande sur laquelle elle agit.

Si *fichier* existait déjà, son contenu est écrasé ; s’il n’existait pas, il est créé.

Remarque 14.2

Avec *zsh* (et *ksh*), le comportement en cas d’existence de *fichier* dépend de la valeur de l’option `NOCLOBBER` (voir 14.1.2). Si cette option est mise, le fichier ne sera pas écrasé et *zsh* affichera un message d’erreur. Dans ce cas, pour indiquer que l’on souhaite écraser un fichier s’il existe déjà, il faut utiliser le signe de redirection “>!”

Exemples 14.1

- (a) `ls -l > fich`
- (b) `cat fich1 fich2 > fich3`

Redirection avec ajout à la fin du fichier de redirection (>>)

Il faut utiliser “>>” au lieu de “>”.

Exemple 14.2

```
ls -l >> fich
```

14.2.2 Redirection du fichier d’erreur (2>)

Le fichier d’erreur peut être redirigé en ajoutant “2>*fichier*” (“2>>*fichier*”, si on ne veut pas écraser *fichier*) à la suite d’une commande (attention, pas d’espace entre le 2 et “>”).

Exemple 14.3

```
find /students -name core -print 2>erreurs
```

14.2.3 Envoi d’un message d’erreur (>&2)

Les messages d’erreur doivent être envoyés vers la voie numéro 2. Voici un exemple (voir 16.4.4) :

```
echo "ERREUR" >&2
```

14.2.4 Redirection de l’entrée standard (<)

L’entrée standard peut être redirigée par “<*fichier*”.

Exemple 14.4

```
mail toto@unice.fr < fichier
```

14.3. PIPE (|)

14.2.5 Pseudo-fichier /dev/null

Le fichier spécial de type “caractères” (voir 1.6.2) `/dev/null` est un pseudo-fichier équivalent à un puits sans fond.

Il est souvent utilisé avec les redirections, en particulier pour ignorer les messages d’erreur :

```
find /students -name core -print 2>/dev/null
```

14.3 Pipe (|)

Les données de sortie d’une commande peuvent être directement utilisées comme données d’entrée d’une autre commande grâce au symbole “|”. En effet, la sortie standard d’une commande placée avant le “|” est redirigée vers l’entrée standard de la commande placée après le “|”.

Les commandes du pipe sont exécutées en parallèle par des processus distincts³. Il faut se le rappeler quand une des commandes du pipe contient un *exit* : la commande *exit* ne fait sortir que du sous-shell qui exécute la commande du pipe et pas du shell dans lequel s’exécute le pipe (voir 17.4.7).

Le code retour du pipe est le code retour de la dernière commande exécutée dans le pipe.

Exemples 14.5

- (a) `ls -l /bin | grep '^d'`
- (b) On peut composer plusieurs pipes :
`ls -l /bin | grep '^d' | lpr`

14.4 Regroupement des commandes

Le regroupement de commandes est souvent utilisé pour appliquer une redirection ou un pipe à plusieurs commandes.

14.4.1 Regroupement entre parenthèses

Les commandes regroupées entre des parenthèses sont exécutées dans un sous-shell avec un environnement différent de celui précédant et suivant les parenthèses.

Exemple 14.6

```
$ pwd
/users/students/toto
```

³. Dans les dernières versions de *zsh*, la dernière commande s’exécute dans le shell dans lequel le pipe a été lancé

```
$ (cd /bin ; pwd)
/bin
$ pwd
/users/students/toto
```

14.4.2 Regroupement entre accolades

Les commandes regroupées entre des accolades ne sont pas exécutées dans un sous-shell comme avec les parenthèses.

Les accolades doivent être séparées des commandes par au moins un espace.

Attention, “}” et “{” ne sont reconnus qu’en début de commande. Ils doivent donc être précédés par un “;” (séparateur de commandes) si on veut les placer sur la même ligne que ce qui précède. L’oubli du “;” est fréquent et il est difficile à repérer car le shell n’indique pas clairement le problème dans ses messages d’erreur.

Exemple 14.7

```
{ echo "Fichiers fich1 et fich2 :" ; pr fich1 fich2 ; } | lpr -h
```

14.5 Processus en arrière-plan (&)

Lorsqu’on lance une commande, le shell attend la fin de l’exécution de la commande avant d’exécuter une autre commande.

Si on ajoute le signe “&” derrière le nom de la commande (précédé ou non d’un espace), le shell n’attend plus la fin de son exécution. On dit qu’il lance la commande en arrière-plan.

Exemple 14.8

```
ls -l | grep '^d' > fich &
```

Quand on lance une commande en arrière-plan dans un shell interactif (pas dans un shellsript), le shell renvoie à l’écran le numéro de “job” entre crochets (le premier processus lancé en arrière-plan par le shell en cours a le numéro 1), et le *pid* de processus (voir 2.2) attribué à cette commande.

On ne peut interrompre une commande lancée en arrière-plan qu’avec la commande *kill* (étudiée en 7.2).

Remarque 14.3

Lorsque l’utilisateur sort du système (en fait, du shell lancé au démarrage), tous les processus lancés en arrière-plan sont tués.

Pour être précis, il existe une commande *nohup* (pas étudiée ici) qui permet de lancer des commandes en arrière-plan, qui continueront leur exécution après la sortie de l’utilisateur.

Les commandes *at* et *batch* permettent de lancer en arrière-plan des commandes à des moments choisis ou lorsque le système n’est pas trop chargé, même si l’utilisateur n’est pas connecté.

14.5.1 Gestion des “jobs” par ksh ou zsh

En ksh ou zsh (mais pas sous sh) l’utilisateur dispose de plusieurs commandes pour choisir le “plan” où les processus lancés s’exécutent :

- l’utilisateur peut faire afficher la liste de ses processus avec leur “numéro de job” pour le shell en cours et leur *pid* (voir 2.2), en tapant

```
jobs [-l]
(l’option -l affiche l’identificateur de processus)
```

- l’utilisateur peut suspendre le processus en cours en tapant

```
[Ctrl] Z
```

- il peut ensuite relancer l’exécution de ce processus en arrière-plan en tapant

```
bg [%n]
n est le numéro de “job” pour le shell (voir 14.5) ; par défaut c’est le processus qui vient d’être suspendu qui est passé en arrière-plan.
```

- les processus en arrière-plan ou suspendus peuvent être passés en avant-plan en tapant

```
fg [%n]
(si aucun numéro de “job” n n’est donné, la commande lance en avant-plan le dernier processus lancé en arrière-plan).
```

Ceci permet de suspendre momentanément un processus pour effectuer un travail ou de revenir ensuite à ce processus exactement à l’endroit où on l’avait laissé. On peut par exemple écrire un programme sous emacs, enregistrer ce programme, suspendre emacs par [Ctrl] Z, compiler et exécuter le programme et revenir sous emacs par fg. Cette possibilité perd de son utilité si l’on travaille en environnement X Window car on peut alors facilement passer d’une application à l’autre sans devoir en suspendre une.

14.6 Alias

Un alias est un synonyme pour une chaîne de caractères dans une commande. Les alias sont particuliers à *ksh* et à *zsh* ; il n’y a pas d’alias dans le shell *sh*.

Par exemple, “alias x='chmod u+x'” définit l’alias “x” : si le premier mot d’une commande est “x”, le shell remplacera ce mot par “chmod u+x”.

On peut défaire un alias par la commande **unalias**, par exemple, “unalias x”.

La liste des alias peut s’afficher en tapant “alias” sans argument. “alias x” affiche le valeur de l’alias x.

Un alias peut être utilisé comme raccourci d’une commande avec ses options comme pour l’alias “x” ci-dessus. On peut aussi l’utiliser pour désigner une commande qui n’est pas placée dans un des répertoires de la variable PATH. Par exemple (sous Unix OSF),

```
“alias quota=/usr/sbin/quotd”
```

14.7 Substitution de commande

Le shell interprète ‘commande’ en le remplaçant par tout ce que la commande envoie sur la sortie standard.

Exemples 14.9

- (a) `date='date'`
- (b) `nom_maj='echo $nom | tr 'a-z' 'A-Z'`
- (c) `echo je suis sous 'pwd'`

Remarques 14.4

- (a) le code retour d’une affectation “a=‘commande’” est le code retour renvoyé par “commande”. On peut ainsi écrire

```
if ligne='grep mot fichier' ; then
# cas ou grep a trouve une ligne
else
# cas ou grep n'a pas trouve une ligne
fi
```

- (b) la commande est exécutée dans un sous-shell.
- (c) Attention, si la commande renvoie plusieurs lignes vers la sortie standard, le comportement est différent selon que l’on travaille sous *sh* et *ksh* ou *zsh*. Par exemple,


```
a='ls -l'
```

 rangera dans a une seule longue ligne sous *sh* et *ksh*, ce qui peut être gênant, mais donnera bien plusieurs lignes sous *zsh*.
- (d) l’interprétation à l’intérieur des ‘ est complexe et peut donner des résultats surprenants :

```
echo \\$HOME
affiche \users/profs/toto, et pourtant

echo 'echo \\$HOME'
affiche $HOME
```

A cause de ce type d’exemple, *ksh* et *zsh* ont ajouté le nouveau type de substitution de commande **\$(commande)** qui n’a pas de telles anomalies :

```
echo $(echo \\$HOME)
affiche \users/profs/toto
```

14.8 Mécanismes d’interprétation

Cette section décrit comment les lignes de commandes sont interprétées par le shell avant que la commande ne soit lancée.

Voici les tâches effectuées par le shell dans l’ordre de leur exécution :

1. interprétation des espaces (pour distinguer les différents mots),
2. substitution de tilde (~) (seulement pour *ksh* ou *zsh*),
3. substitution des alias (seulement pour *ksh* ou *zsh*),
4. interprétation des redirections, des pipes⁴ et des regroupements de commandes,
5. interprétation des affectations⁵,
6. substitution des paramètres de position et de variables (\$),
7. substitution des commandes,
8. génération des noms de fichiers (caractères * ? [] ; voir 4.5.1).

14.9 Inhiber l’interprétation du shell

On peut inhiber l’interprétation du shell (on dit aussi dépersonnaliser) avec les caractères suivants :

- \ inhibe l’interprétation spéciale du caractère suivant.
- ' toute interprétation est inhibée pour les portions de lignes entourées de '
- " dans les portions de lignes entourées de " (guillemets), l’interprétation des espaces, la génération des noms, la substitution des alias et de ~ sont inhibées mais pas la substitution des paramètres (\$). Une apostrophe n’est pas interprétée entre deux guillemets (ce qui permet d’introduire le caractère ' dans une chaîne de caractères.

Exemple 14.10

Essayez cette suite de commandes :

```
a=bonjour
echo $a bc *
echo \$a bc
echo "\$a bc *"
echo '$a *'
```

4. il n’y a pas de génération des noms de fichiers dans l’interprétation des parties correspondant aux fichiers dans les redirections

5. comme pour les redirections, les étapes suivantes sont aussi traitées à part pour les affectations ; par exemple, aucune génération des noms n’est effectuée dans “a=*”

14.10 Recherche d’une commande par le shell

Lorsque l’utilisateur tape une commande au clavier, voici les différentes étapes qui conduisent à l’exécution de la commande :

1. Interprétation de la ligne de commande par le shell,
2. Recherche de la commande,
3. Exécution de la commande avec redirection préalable des entrées-sorties par le shell si nécessaire.

Voici comment s’effectue la recherche de la commande :

- Si la commande correspond à une commande interne du shell, cette commande est exécutée à l’intérieur du shell,
- sinon, si la commande correspond à une fonction (voir 16.3), cette fonction est exécutée à l’intérieur du shell,
- sinon il ne peut s’agir que d’une commande correspondant à un fichier de l’arborescence. Si le nom de la commande ne comporte pas de “/”, la variable PATH est examinée pour rechercher le répertoire où se trouve la commande.
 - Si le fichier est un shellscript un nouveau shell est créé ; il lit les lignes du shellscript et les exécute,
 - si le fichier contient du code binaire exécutable, un nouveau processus est créé, qui exécute ce code.

14.11 Lancement de l’exécution d’un shellscript

Un shellscript est un fichier qui contient des noms de commandes et des instructions internes au shell. On peut utiliser un shellscript comme toutes les autres commandes. En particulier un autre shellscript peut l’utiliser. On peut ainsi se construire facilement de nouvelles commandes adaptées à sa propre façon de travailler.

Les instructions internes du shell forment un langage de haut niveau possédant des instructions de tests et de boucles, des variables et des fonctions.

On peut lancer l’exécution d’un shellscript de trois manières différentes.

14.11.1 Lancement par le nom du shellscript

Si on a l’autorisation de lecture et d’exécution sur le shellscript, on peut le lancer en tapant simplement son nom.

Le système s’aperçoit que le fichier qui contient le shellscript n’est pas écrit dans du code directement exécutable par le processeur de la machine et il lance donc un interpréteur pour lire et exécuter les lignes contenues dans le fichier.

Toutes les commandes du shellscript sont exécutées comme si elles étaient tapées au clavier. Une différence importante est que les commandes sont exécutées dans un processus fils (dont la zone de code contient le code de l’interpréteur) qui lit les lignes du shellscript et lance leur exécution. Si le shellscript modifie l’environnement de départ, celui-ci est rétabli lorsque l’exécution du shellscript est terminée (voir 15.2.9).

Il est important de noter que, sauf cas particulier étudié ci-dessous (`#!`),

- si on travaille sous *sh* ou *ksh*, l’interpréteur lancé implicitement pour lire et exécuter le shellscript est le shell courant (*sh* ou *ksh* selon le cas),
- si on travaille sous *zsh*, l’interpréteur lancé implicitement est *sh* et pas *zsh* ! Si le fichier contient des instructions comprises par *zsh* et pas par *sh*, il y aura une erreur à l’exécution.

Pour que le shellscript soit interprété par *zsh*, il doit commencer par la ligne :

```
#! /usr/local/bin/zsh -f
```

Exécution d’un fichier dont la première ligne commence par “#!”

Si le fichier comporte une première ligne qui commence par “#!”, Unix lance comme interpréteur non pas le shell habituel, mais le programme dont le nom absolu suit “#!” (avec éventuellement un argument comme “-f” dans l’exemple ci-dessus), auquel il applique comme arguments la ligne de commande tapée par l’utilisateur. Voici des exemples concrets :

Si l’utilisateur tape la commande “`c1 f1 f2`”,

- si `c1` commence par “`#! /bin/sh`”, la commande exécutée sera “`/bin/sh c1 f1 f2`”. Le shellscript “`c1`” est lu par le shell *sh* et exécuté avec les arguments “`f1 f2`”.

- si `c1` commence par “`#! /usr/local/bin/zsh -f`”, la commande exécutée sera “`/usr/local/bin/zsh -f c1 f1 f2`”. Le shellscript “`c1`” est lu par le shell *zsh* et exécuté avec les arguments “`f1 f2`”. L’option “`-f`” indique que le fichier d’initialisation “`.zshenv`” ne doit pas être lu ; voir 14.11.3.

- si `c1` commence par “`#! /bin/awk -f`”, la commande exécutée sera “`/bin/awk -f c1 f1 f2`”. Le programme “`c1`” (nécessairement écrit avec du code “`awk`”) est interprété par *awk* sur les fichiers “`f1`” et “`f2`” (voir 13.10).

Remarque 14.5

Ce mécanisme ne peut fonctionner qu’avec les interpréteurs (tels les shells ou *awk*) pour lesquels “`#`” introduit des commentaires.

14.11.2 Lancement par l’appel de la commande interne “.”

Il reste une autre possibilité pour faire exécuter un shellscript. Le nom du shellscript est précédé d’un point (laisser un espace entre le point et le nom du shellscript) :

```
. fichier
```

Dans ce cas, le shellscript *fichier* est exécuté comme si ses commandes étaient tapées directement au clavier. Un nouveau processus n’est pas créé. Les modifications de l’environnement seront donc conservées.

En fait, “.” est une commande interne du shell qui lit toutes les commandes contenues dans le fichier et les exécute comme si elles avaient été tapées au clavier.

Remarque 14.6

Quel que soit le shell, le shellscript est toujours lu et exécuté par le shell en cours, à la différence de l’appel par le nom du shellscript où il y a une différence entre *zsh* et les autres shells (voir remarques de 14.11.1).

Exemple 14.11

Soit le shellscript *cdbin* contenant la seule commande

```
cd bin
```

Si on tape

```
cdbin
```

(le nom du shellscript), le répertoire sera le même après l’exécution qu’avant. En effet, le shellscript s’exécute dans son propre environnement et il n’a pas d’action sur l’environnement du shell de départ.

Si on tape

```
. cdbin
```

le répertoire courant sera *bin* après l’exécution car *cd* est une commande interne du shell et non pas une commande externe correspondant à un fichier Unix. Elle est donc exécutée directement par le shell sans faire appel à un autre shell qui aurait son propre environnement. Puisque l’on a exécuté *cdbin* avec un “.” initial, les commandes de *cdbin* sont donc exécutées par le shell initial dont le répertoire courant devient donc le répertoire *bin*.

14.11.3 Lancement par l’appel explicite d’un shell

Si on a l’autorisation de lecture sur le shellscript, on peut lancer son exécution par :

```
zsh [-f] fichier
```

(si on travaille avec *zsh*) dans ce cas, un nouveau shell est lancé explicitement. Celui-ci lit les commandes du shellscript *fichier* et les fait exécuter comme si les commandes avaient été tapées au clavier.

Le fichier *.zshenv* est exécuté sauf si le shell est lancé avec l’option “-f”. Le fichier */etc/zshenv* est toujours exécuté.

Il faut noter que c’est le mode de lancement de *zsh* lorsqu’on lance une commande “*rsh machine commande*” sur une machine distante et un compte utilisateur qui a *zsh* comme shell de démarrage (voir cours sur les réseaux).

14.12 Lancement explicite d’un shell

```
zsh [-f]
```

Les fichiers *.zshenv*, *.zshrc* et */etc/zshrc* sont exécutés sauf si le shell est lancé avec l’option “-f”. Le fichier */etc/zshenv* est toujours exécuté.

Il faut noter que c’est le mode de lancement de *zsh* lorsque l’on ouvre une fenêtre avec *xterm*.

Chapitre 15

Variables, environnement

15.1 Paramètres, variables de position

15.1.1 Paramètres des shellscrip

Lorsqu'on lance une commande, on peut la faire suivre par des arguments ou paramètres.

À l'intérieur d'un shellscrip, ces paramètres sont désignés par les variables de position \$1, \$2,..., \$9 (\$0 désigne le nom de la commande).

Exemples 15.1

- (a) Si le shellscrip lt contient

```
ls -lt $1 | more
on peut lancer

lt /users/students/jean
```

- (b) Si le shellscrip "cherche" contient

```
find $1 -name "$2" -print
on peut taper

cherche /usr "compt*"
qui cherchera sous l'arborescence du répertoire /usr les fichiers dont le nom commence
par compt.
```

15.1.2 Donner des valeurs aux paramètres de position (set)

La commande set permet d'affecter des valeurs aux paramètres de position à l'intérieur d'un shellscrip. La commande suivante donne les valeurs val1, val2, ... aux paramètres \$1, \$2, ... :

```
set val1 val2 ...
```

15.2. VARIABLES

Cette possibilité est souvent utilisée dans les shellscrips pour récupérer un ou plusieurs mots d'une ligne. Par exemple, les deux commandes suivantes affichent le mois et l'année de la date système :

```
set `date`
echo $2 $6
```

Remarque 15.1

```
set -x abc
```

ne convient pas pour donner à \$1 la valeur "-x" et à \$2 la valeur "abc". En effet, set veut penser que "-x" est l'option du shell étudiée en 17.11. L'option "--" indique que les paramètres suivants ne sont pas des options du shell mais les valeurs des paramètres de position. Il faut donc écrire :

```
set -- -x abc
```

15.2 Variables

15.2.1 Identificateur

Durant une session de travail d'un shell (délimitée par le démarrage et l'arrêt du shell) on peut utiliser des variables désignées par un identificateur. Cet identificateur peut contenir des lettres, des chiffres ou le caractère souligné ; il doit commencer par une lettre ou le caractère souligné.

15.2.2 Affectation

On peut affecter une chaîne de caractères aux variables :

```
variable=valeur
```

(pas d'espaces de part et d'autre du signe =!).

"variable=" affecte la chaîne vide à *variable*.

Remarque 15.2

Le code retour d'une affectation du type

```
a='commande'
```

est le code retour renvoyée par la *commande*.

On peut par exemple écrire

```
if lignes=`grep mot fichier` ; then ...
et ce qui suit le "then" ne sera exécuté que si le fichier contient le mot.
```

15.2.3 Désignation de la valeur de la variable (\$)

On peut réutiliser la valeur affectée à une variable dans toute commande.

`$variable`

désigne la valeur de *variable*. C’est le shell qui fera le remplacement.

Remarque 15.3

Le shell ne fera pas le remplacement dans les portions de commande placées entre apostrophes ou si le “\$” est précédé d’un “\”. Au contraire de la génération de noms de fichiers, le remplacement est effectué dans les portions de commande placées entre guillemets (voir 14.8).

Exemple 15.2

```
r1=us2/acct/jean/dvp
r2=us2/bin
cd $r1
cp fich $r2
```

Si la valeur de la variable doit être suivie d’un caractère autorisé dans un nom de variable, on entoure d’accolades le nom de la variable :

Exemple 15.3

```
serie=math
cp classe/${serie}1 rep
```

15.2.4 Décomposition en mots des valeurs des variables de zsh

Attention ! *zsh* ne découpe pas la valeur d’une variable en mots. Le programme suivant va copier les deux fichiers “f1” et “f2” sous le répertoire “rep” s’il est exécuté sous *sh* ou *ksh*, mais il ne fonctionnera pas sous *zsh* car *cp* essaiera de copier le fichier de nom “f1 f2” (un seul nom avec un espace au milieu) :

```
fichiers="f1 f2"
cp $fichiers repertoire
```

Il faut écrire “\${fichiers}” si on veut que la valeur de la variable “fichier” soit découpée en mots. Sous *zsh* la commande de l’exemple précédent doit donc s’écrire :

```
cp ${fichiers} repertoire
```

15.2.5 Supprimer une variable (unset)

`unset variable`

Remarque 15.4

La commande *unset* n’existe pas dans le shell *sh*.

15.2.6 Variables spéciales du shell

Voici les principales variables utilisées par le shell.

HOME répertoire de login

PATH répertoire de recherche des commandes

Exemple : PATH=/bin:/usr/bin:/users/students/util:.

Le répertoire courant peut être indiqué par “.” au début ou à la fin de la valeur de PATH. Il est recommandé de ne pas mettre trop de répertoires dans cette variable pour ne pas surcharger le système en cas de recherche d’une commande qui n’existe pas (et qui sera recherchée dans tous ces répertoires).

CDPATH

est une variable quelquefois utile si on travaille souvent dans des répertoires dont le chemin est long à taper. Si l’utilisateur tape un nom qui ne commence pas par un “/” en argument de *cd*, *cd* recherchera ce nom dans les répertoires indiqués dans CDPATH. Le format de CDPATH est le même que celui de PATH.

Attention, sous *sh* et *ksh*, si la variable CDPATH a une valeur, cette variable est utilisée exclusivement pour rechercher les répertoires dont le nom ne commence par “/” dans une commande “*cd*” (comme pour la variable PATH et la recherche de commandes). Si on est par exemple placé dans son répertoire HOME et que celui-ci contient un répertoire bin, la commande “*cd bin*” déplacera dans le premier répertoire bin placé sous un des répertoires de la variable CDPATH, et renverra une erreur s’il n’y en a pas. Veuillez donc à toujours avoir le répertoire courant au début de la valeur de CDPATH si vous ne voulez pas avoir ce comportement. Sous *zsh*, le répertoire courant est toujours cherché en premier même s’il n’est pas dans la variable CDPATH.

PS1

premier prompt Unix. *zsh* offre de nombreuses possibilités particulières pour personnaliser le prompt (voir 15.3).

PS2

deuxième prompt Unix (> par défaut)

TERM

nom du type de terminal utilisé

IFS

séparateur(s) de mots dans les commandes (espace, tabulation ou passage à la ligne par défaut). IFS est utilisée lors d’une substitution de commande (‘...’ ou \$()), de paramètre ou de variable (\$), ou lors de la lecture d’une ligne par la commande interne *read*, pour découper une ligne en mots. On remarquera que IFS n’est pas utilisée entre guillemets car les guillemets inhibent la séparation en mots. Essayez la suite de commandes suivante :

```
$ IFS=/
$ read x y
v1/v2
$ echo $x
$ var=a/b/c
$ echo $var # $=var avec zsh
$ echo "$var" # différent si entre guillemets
$ cat /etc/passwd
“IFS=” remet la valeur par défaut.
```

Les variables suivantes ont leur valeur donnée automatiquement par le shell ; on ne peut modifier leur valeur.

#	nombre de paramètres de position
?	valeur retournée par la dernière commande exécutée
!	numéro de processus de la dernière commande lancée en arrière plan
\$	numéro de processus du processus en cours
*	\$* est équivalent à \$1 \$2 ...
@	\$@ est équivalent à \$1 \$2 ... comme \$*; mais il existe une différence subtile entre les 2 variables * et @: "\$*" (entouré de guillemets) représente un seul mot composé des arguments \$1 \$2 ... séparés par un espace alors que "\$@" représente différents mots égaux aux contenus de \$1, \$2, ..., avec un espace séparateur qui sera interprété par le shell comme un séparateur de mots (voir exemple de l'instruction "for" en 17.4.3).

15.2.7 Afficher la valeur d'une variable (echo)

```
echo [-n] $variable
```

Plus généralement, echo envoie sur la sortie standard les valeurs de tous ses paramètres, séparés par un espace. Dans la commande ci-dessus, c'est le shell qui va remplacer *\$variable* par la valeur de la *variable* et echo affichera cette valeur.

En Unix BSD, l'option `-n` permet d'éviter le passage à la ligne après l'affichage. En Unix Système V, on peut passer en paramètres de *echo* des caractères spéciaux et c'est le caractère `"\c"` qui indique que l'on ne doit pas passer à la ligne (ne pas oublier de protéger `"\"` de l'interprétation du shell). Par exemple :

```
echo "Donnez le nom du fichier \c: "
```

Exemples 15.4

(a) `echo valeur de HOME : $HOME`

(b) `echo "Voici 5 espaces entre parentheses : ()"`

Question : que se passe-t-il si on enlève les guillemets ?

La commande interne *print* de *ksh* et de *zsh*, permet des formats d'affichage plus sophistiqués, mais cette commande n'existe pas dans le shell *sh*.

15.2.8 Entrée de la valeur d'une variable au clavier (read)

```
read variable...
```

Une ligne est lue sur l'entrée standard. Le premier mot est affecté à la première variable, le deuxième mot est affecté à la deuxième variable, ... La dernière variable reçoit le restant de la ligne.

S'il y a moins de mots que de variables, les dernières variables sont initialisées à la chaîne vide.

Le code retour est 0 sauf si on a atteint une fin de fichier.

Exemples 15.5

(a) `echo Nom du repertoire a lister :`

```
read rep
ls -ld $rep | more
```

pour lire au clavier le nom d'un répertoire à lister.

(b) `read v1 v2 v3 <fichier`

lit dans la première ligne du fichier les valeurs des variables *v1*, *v2* et *v3*. On verra la page 131 comment lire toutes les lignes d'un fichier.

15.2.9 Portée d'une variable (export), environnement de travail

L'environnement de travail d'un shell est constitué des variables auxquelles le shell a donné une valeur et des variables que le shell a héritées du processus qui l'a lancé.

Une variable n'est ordinairement utilisée que dans le shell où elle reçoit son affectation.

Si on veut la réutiliser dans les programmes appelés ultérieurement par le shell, on doit exporter la valeur par export.

Remarque 15.5

On dira que les programmes appelés ultérieurement par un shellscript sont des *descendants* du shellscript qui les a appelés, ou que le shellscript est un *ancêtre* de ces programmes.

```
export variables...
```

recopie la valeur des variables dans l'environnement qui sera passé aux programmes appelés ultérieurement.

Remarques 15.6

(a) Puisque c'est une recopie, les éventuelles modifications des variables exportées n'ont pas de conséquence : elles ne modifieront pas l'environnement du shellscript appelant.

(b) *ksh* et *zsh* permettent d'affecter des valeurs à des variables tout en exportant les variables :

```
export a=3 fichier=fich12
```

On obtiendra une erreur si le shellscript est exécuté par *sh* et il faut donc l'éviter.

(c) `export` qui ne se comporte pas de la même façon dans les tous les shells :

- avec *sh*, si un programme modifie une variable qui a été exportée par un programme ancêtre et s'il n'exporte pas cette variable (avant ou après sa modification), les programmes descendants qu'il appellera, recevront la valeur initiale pour cette variable (et non la valeur modifiée).

Si un programme modifie la valeur d'une variable et veut transmettre la nouvelle valeur aux programmes qu'il appelle, il doit donc exporter cette variable.

- avec *ksh* et *zsh*, si une variable a été exportée par un programme, elle le sera automatiquement par tous les programmes appelés par le programme.

15.2.10 Visualisation des variables disponibles (set, printenv)

`printenv`

affiche les valeurs des variables de l'environnement du processus en cours (celles qu'il a recues et celles qu'il a exportées); "export" (sans nom de variable) donne un résultat semblable sous *zsh* et *ksh*, mais sous *sh*, "export" donne uniquement le nom des variables exportées par le shell en cours.

`set`

affiche les valeurs des variables disponibles (même celles qui ne sont pas exportées et qui n'appartiennent pas à l'environnement qui sera transmis).

15.3 Personnalisation de l'environnement

15.3.1 Options des shells (set, setopt)

De nombreuses options permettent de configurer le fonctionnement des shells (voir en particulier 14.1.2). Ces options peuvent être entrées en paramètres lors du lancement du shell ou positionnées grâce à la commande *set*. On se reportera au manuel de référence du shell pour la syntaxe utilisée pour positionner les options.

On a vu aussi en 3.3 l'option "`-o ignoreeof`" et on verra l'utilité des options "`-v`" et "`-x`" pour la mise au point des shells en 17.11.

La fonction interne *setopt* de *zsh* permet aussi de positionner les options du shell. On peut avoir une liste de toutes les options positionnée en tapant "`setopt`" seul.

Dans les sections suivantes nous allons étudier les différents fichiers de configuration de l'environnement que le shell zsh exécute automatiquement au démarrage.

15.3.2 Fichiers de personnalisation

À chaque fois qu'un shell *zsh* est lancé, des fichiers de configuration sont lus et exécutés automatiquement par le shell.

Certains fichiers sont communs à tous les utilisateurs et permettent à l'administrateur du système de leur donner un environnement minimum commun. Ce sont les fichiers *zshenv*, *zshrc*, *zprofile* et *zlogin* du répertoire */etc*.

Les autres fichiers sont situés dans le répertoire HOME de chaque utilisateur. Celui-ci peut les créer ou les modifier à sa convenance. Ce sont les fichiers *.zshenv*, *.zshrc*, *.zprofile* et *.zlogin*.

Ces fichiers ne sont pas toujours tous exécutés. Voyons les différents cas.

1. */etc/zshenv*

15.3. PERSONNALISATION DE L'ENVIRONNEMENT

est le seul fichier qui est toujours exécuté. Si nécessaire, il donne un environnement minimum pour travailler.

Si zsh a été lancé avec l'option "`-f`", c'est le seul fichier de configuration a été exécuté.

2. *.zshenv*

est toujours exécuté si *zsh* n'a été pas lancé avec l'option "`-f`". Il initialise les variables d'environnement indispensables au bon fonctionnement du shell dans tous les cas (PATH par exemple).

3. */etc/zprofile* et *.zprofile*

(exécutés par les shells de login seulement) initialisent les variables d'environnement indispensables au bon fonctionnement du shell juste après un login. Ce fichier n'est en général pas utilisé car un shell de login est le plus souvent un shell interactif et utilise alors plutôt le fichier "*.zlogin*" (voir ci-dessous).

4. */etc/zshrc* et *.zshrc*

(exécuté par les shells interactifs seulement) contiennent les définitions d'alias, de prompts et des variables liées à l'utilisation interactive d'un shell.

Remarque 15.7

Quand on ouvre une nouvelle fenêtre X Window, ces fichiers sont exécutés (après les fichiers */etc/zshenv* et *.zshenv*) par le nouveau shell qui s'exécute dans la fenêtre.

Exemple 15.6

```
umask 022
# definition des alias
alias ll='ls -l'
alias x='chmod u+x'
# definition prompt : ordinateur et repertoire courant
PS1=%m'(%~) '
# definition de variables
export PRINTER=17
export MAIL=/usr/spool/mail/$USER
export MANPATH=/usr/man:/usr/local/man:/usr/local/X11R5
export SAVEHIST=300 HISTFILE=$HOME/.sh_history HISTSIZE=300
# Pour compatibilite avec ksh :
setopt NO_NOMATCH
export NULLCMD=:
```

5. */etc/zlogin* et *.zlogin*

(exécutés par les shells de login interactifs seulement) contiennent les initialisations qui doivent être faites au début d'une session de travail interactive, en particulier l'affectation des variables `TERM` et `DISPLAY`.

Exemple 15.7

```
if tty -s ; then
    stty dec crt
fi
if [ -z "$DISPLAY" ] ; then      # cas où DISPLAY est vide
    echo '-----'
    echo '  1. vt100
2. X Window'
    echo -n "Type de votre terminal (2 par défaut): "
    read terminal
    case $terminal:-2 in
        1) TERM=vt100;;
        2) TERM=xterm
            echo "Nom du DISPLAY ? "
            read DISPLAY;;
    esac
fi
```

Remarque 15.8

Les fichiers utilisés lors de l'écriture de fichiers de configuration (au moins ceux qui ne sont pas liés à un shell de login) doivent être désignés par leur nom absolu car les shells peuvent être lancés d'un autre répertoire que le répertoire de login.

15.4 Stratégie pour la personnalisation de l'environnement

On travaille le plus souvent dans plusieurs domaines bien distincts, par exemple, un environnement pour le développement en langage Java, un environnement pour le travail avec une base de données et un autre pour la recherche d'information sur le Web.

Dans ce cas, il est préférable d'avoir des environnements de travail différents plutôt que d'avoir un seul environnement alourdi par un grand nombre de variables et avec une variable `PATH` contenant de nombreux répertoires.

Pour cela le plus simple est d'alléger les fichiers d'initialisation vus ci-dessus en ne gardant que les éléments qui seront utiles à tous les environnements, et de créer un shellscript d'initialisation pour chacun des environnements. Chacun de ces shellscripts initialisera les variables (en particulier la variable `PATH`), placera l'utilisateur dans le bon répertoire de travail et lancera les programmes nécessaires (par exemple, il ouvrira une nouvelle fenêtre X Window dans laquelle il lancera un programme pour travailler avec une base de données).

Chapitre 16

Compléments sur le shell

Ce chapitre donne quelques compléments sur des notions déjà étudiées dans les deux chapitres précédents. Il présente aussi quelques nouveaux aspects moins essentiels du langage du shell.

16.1 Listes de commandes

Un pipe est une suite de commandes séparées par `|`. Le code retour d'un pipe est le code retour de la dernière commande exécutée.

Une liste de commandes est une suite de pipes ou de simples commandes ; dans une suite, par commodité, "pipe" pourra désigner une simple commande) séparées par un des signes suivant :

- `;` correspond à une exécution séquentielle des pipes
- `&` correspond à une exécution séquentielle des pipes sans attente de la fin de l'exécution du pipe précédent
- `&&` le pipe suivant n'est exécuté que si le pipe précédent renvoie la valeur 0 (déroutement "normal" du pipe précédent)
- `||` le pipe suivant n'est exécuté que si le pipe précédent ne renvoie pas la valeur 0 (déroutement "anormal" du pipe précédent)

Exemples 16.1

(a) `rm fich || echo Suppression impossible !!`

(b) `cat $2 | { write $1 || mail $1 ;}`
shellscrip qui envoie un message sur l'écran ou dans la boîte aux lettres d'un utilisateur connecté à la machine locale.

Quelques subtilités (**false** est une commande qui ne fait que renvoyer un code retour non nul ; **true** est une commande qui ne fait que renvoyer un code retour nul) :

`false && echo 1 && echo 2`
n'affiche rien

```

true && echo 1 && echo 2
affiche 1 et 2
false || echo 1 && echo 2
affiche 1 et 2
true || echo 1 && echo 2
affiche 2

```

16.2 Ordre de priorité

L'ordre de priorité des différents signes qui ont une signification particulière pour le shell est (par ordre décroissant) :

1. les redirections
2. |
3. && et ||
4. & et ;

16.3 Fonctions

Les shellsripts peuvent contenir des définitions de fonctions qui pourront ensuite être utilisées dans le shell dans lequel elles ont été définies.

Les fonctions sont plus rapidement exécutées que les shellscripts puisque le shell n'a pas à aller chercher de fichier pour l'exécuter. De plus une fonction peut modifier les variables du shell appelant puisqu'elle s'exécute dans le même environnement que ce shell.

Définition d'une fonction

```

nom-fonction ()
{
...
...
}

```

L'accolade de fin de fonction doit être en début de ligne (ou précédée de;).

Appel de la fonction

```

nom-fonction param1 param2 ...

```

Exécution de la fonction

Les commandes de la fonction s'exécutent dans le shell dans lequel la fonction a été appelée. Les \$1, \$2,... sont remplacés par les paramètres d'appel de la fonction (*param1 param2...*).

Exemple 16.2

Si la fonction lt est définie par :

```
lt () { ls -lt $1 | more }
```

on pourra lancer la fonction par :

```
lt /bin
```

Placée à l'intérieur d'une fonction la commande

```
return n
```

joue le rôle de la commande exit pour un shellsript : elle arrête l'exécution de la fonction en renvoyant le code retour *n*.

16.4 Compléments sur les redirections

16.4.1 Redirection de l'entrée standard sur fichier inclus (<<)

“<<” permet d'indiquer dans un programme des lignes qui seront lues sur l'entrée standard pendant l'exécution du programme. Le délimiteur de ces lignes est indiqué juste après le signe “<<”. Le délimiteur de fin doit absolument être le premier caractère dans sa ligne (avec éventuellement des tabulations si on fait suivre “<<” de “-” ; voir ci-dessous) sinon on obtient des messages du type “fin de fichier inattendue” car le délimiteur n'a pas été reconnu.

L'ensemble des lignes ainsi délimitées est appelé un fichier inclus. En effet, tout se passe comme si elles appartenaient à un fichier vers lequel on redirigeait l'entrée standard.

Si le premier caractère du délimiteur est précédé de “\”, cela signifie que le shell ne doit pas interpréter les caractères spéciaux (*, ?, ‘, ...) situés entre les deux délimiteurs. Tout se passe comme si le texte entre les deux délimiteurs était entouré du caractère ‘.

Si “-” est ajouté à la suite de “<<”, toutes les tabulations de début de ligne sont enlevées du fichier inclus. Ceci permet d'indenter ces lignes si elles appartiennent, par exemple, à une boucle “for” ou à une structure “if”.

Utilisé avec l'éditeur de texte *ed*, la notion de fichier inclus permet de modifier un fichier avec des commandes construites dans un shellsript. Nous n'étudierons pas ici l'éditeur de texte *ed*. Les commandes de *ed* utilisées dans l'exemple ci-dessous sont :

s substitue une chaîne de caractères à une autre. Dans chaque ligne, seule la première occurrence de la chaîne cherchée sera remplacée sauf si on ajoute l'option “g” à la fin de la commande (voir exemple ci-dessous). Comme la commande suivante, “s” peut

être précédé d'un intervalle de lignes qui indique les lignes sur lesquelles portera la commande (“\$” désigne la dernière ligne).

En fait, les chaînes de caractères peuvent être des expressions régulières et on peut se référer dans la deuxième chaîne à une sous-chaîne de la première. Par exemple,

```
s/^\([_:]*\)\\.(*\\)$/\2 est le prenom de \1/
```

transformera “Dupond,Pierre” en “Pierre est le prenom de Dupond”.

Consultez le manuel en ligne de *ed* pour plus de précisions.

g exécute une commande sur toutes les lignes qui contiennent une expression régulière (par exemple, “1,5**g**/~toto/d” supprime les lignes qui commencent par “toto” dans les 5 premières lignes.

w sauvegarde le fichier en cours d’édition.

q quitte *ed*.

La syntaxe d’appel de *ed* est :

```
ed [-s] fichier
```

où l’option “-s” indique que l’on ne veut pas que *ed* affiche certaines informations comme, par exemple, le nombre d’octet enregistrés lors de la commande “w”.

Exemple 16.3

```
ed -s $3 <<FIN
1,\$s/$1/$2/g
w
q
FIN
```

Le shellscript “remplace” ci-dessus remplace une chaîne par une autre dans un fichier (par exemple: “remplace ch1 ch2 fichier”)

16.4.2 Redirection pour tout un shell (exec)

La commande interne *exec*, outre le lancement d’une commande en remplacement du shell, permet des redirections qui seront valables jusqu’à la sortie du shell. La commande suivante, placée dans un shellscript, enregistrera tous les messages d’erreurs du shellscript dans le fichier “suivi.err”, jusqu’à la fin du shellscript :

```
exec 2>suivi.err
```

16.4.3 Descripteurs de fichier supérieurs à 2

On sait que les trois fichiers standards sont ouverts avec les descripteurs 0, 1 et 2. Un shellscript peut ouvrir d’autres fichiers en utilisant les descripteurs supérieurs à 2.

Si le signe de redirection “<” ou “>” est précédé d’un nombre entier, cela signifie que la redirection concerne le fichier dont le descripteur est le nombre entier donné, et non l’entrée, la sortie ou l’erreur standard. Il ne doit pas y avoir d’espace entre le numéro et le signe qui le suit.

Exemples 16.4

(a) **3< fichier** et **3> fichier**

ces deux commandes ouvrent fichier avec le descripteur 3 (et non pas le descripteur 0 ou 1 s’il n’y avait pas eu le 3). La différence est que la deuxième redirection écrase le fichier.

(b) Pour fermer le fichier ouvert avec le descripteur 3, on utilise les commandes **<&-** et **>&-** (pas d’espace après le signe de redirection) qui ferment les fichiers d’entrée et de sortie standard ; la commande suivante ferme le fichier de descripteur 3 :

```
3<&-
```

16.4.4 Redirection vers un fichier désigné par son descripteur

```
>&n (resp. <&n)
```

signifie que la sortie (resp. entrée) standard est redirigée vers le fichier ouvert avec le descripteur *n* et que les deux fichiers partagent le même pointeur de fichier (voir fonction *dup()* en langage C). Il ne faut pas d’espace entre > et &.

“> **fichier**” écrase **fichier** et positionne le pointeur de positions courante du fichier au début du fichier alors que “>&n” (ou “<&n”) désigne un fichier sans modifier la valeur de position courante.

Exemples 16.5

(a) Pour afficher un message d’erreur :

```
echo "$0: Erreur ..." >&2
```

(b) Si on veut faire imprimer les erreurs de syntaxe d’un programme en langage C :

```
cc pgm.c 2>&1 | lpr
```

car les erreurs sont redirigées vers la sortie standard qui est envoyée par le pipe à la commande lpr.

16.4.5 Ordre d’évaluation des redirections

Les redirections sont évaluées de gauche à droite. Par exemple, les deux redirections suivantes

```
1>fich1 2>&1
```

ont pour effet de rediriger la sortie standard et le fichier standard des erreurs vers fich1. Dans l’ordre inverse, seule la sortie standard aurait été redirigée vers fich1.

16.4.6 Redirection sans commande avec *zsh*

Avec les shells *sh* ou *ksh*, une redirection sans commande ne va rien faire (“< fichier”) ou va créer un fichier vide (“> fichier”). Avec *zsh*, une commande par défaut est ajoutée avant la redirection. Le nom de cette commande est contenue dans la variable `NULLCMD` ; c’est la commande *cat* par défaut. Si on veut que *zsh* se comporte comme *sh* et *ksh*, on ajoute les lignes suivantes dans le fichier `.zshrc` :

```
NULLCMD=:
export NULLCMD
```

16.5 Compléments sur les variables

16.5.1 Valeurs par défaut pour les variables

Il est possible de substituer des valeurs aux valeurs des variables ou des paramètres de position comme suit (les valeurs réelles des variables ou paramètres ne sont pas modifiées par ces substitutions sauf pour “=”).

On dit qu’une variable est activée si elle a reçu une valeur. Un paramètre de position est activé si la commande a été lancée avec au moins autant d’arguments que le numéro du paramètre.

```
${variable-mot}
```

si la variable n’est pas activée, elle est remplacée par la valeur `mot`.

```
${variable?mot}
```

si la variable n’est pas activée, `mot` est affiché et l’exécution est arrêtée. Si `mot` est omis, un message standard est affiché.

```
${variable+mot}
```

si la variable est activée, elle est remplacée par `mot`, sinon la chaîne vide est substituée.

```
${variable=mot}
```

si la variable n’est pas activée, sa valeur devient `mot`. Ne marche pas pour les paramètres de position.

Remarque 16.1

“mot” est interprété par le shell, sauf quand il suit un “?”. Il peut ainsi être une substitution de commande ou désigner la valeur d’une variable.

Exemples 16.6

- (a) `cd ${rep=$HOME}`
- (b) `var=${2?il manque des parametres}`
- (c) `cd ${DIR=$HOME}`

Si les signes `-`, `?`, `+`, `=` sont précédés du signe “:”, le comportement pour une variable égale à la chaîne vide est le même que pour une variable non activée. Par exemple,

```
${var:=val}
```

donnera la valeur “val” à la variable `var` si celle-ci est égale à la chaîne vide.

16.5.2 Modifier l’environnement d’une commande

On peut modifier l’environnement d’une commande en la préfixant d’une ou de plusieurs affectations de variables (séparées par des espaces).

Exemple 16.7

```
rep=/usr/bin commande arguments...
```

Attention, la portée de cette affectation se limite à la commande préfixée ; l’affectation ne modifiera pas l’environnement du shell dans lequel la commande aura été tapée.

Exemple 16.8

```
a=5
a=6 echo $a
```

affichera 5 et pas 6 car la commande `echo` est une commande interne au shell.

Erreur à ne pas faire : si l’on veut concaténer deux valeurs dans une variable (par exemple “abc” et “def”), en les séparant par un espace, ne pas écrire

```
a=abc def
```

qui serait interprété comme le lancement de la commande “def” en lui exportant la valeur “abc” de la variable “a”. Ecrire plutôt :

```
a="abc def"
```

16.5.3 Facilités de *ksh* et *zsh* pour le traitement des valeurs des variables

ksh et *zsh* offrent un plus par rapport à *sh* (à ne pas utiliser si on veut écrire un shellscript portable sur *sh* ; utiliser plutôt la commande *expr*) :

```
${#variable}
```

donne la taille du contenu de la *variable*.

```
${variable#modèle}
```

(où *modèle* est une chaîne de caractères pouvant contenir les caractères spéciaux utilisés par la génération des noms de fichiers par le shell) si *modèle* correspond au début de la valeur de *variable*, la valeur renvoyée est la valeur de *variable* sans la plus petite portion de texte du début qui correspond à *modèle*. Sinon, la valeur renvoyée est celle de *variable*.

```
${variable##modèle}
```

la même chose que ci-dessus mais on supprime la plus grande portion de texte du début qui correspond à *modèle*.

```
${variable%modèle}
```

ou

`${variable%%modèle}`

la même chose que # et ## mais on supprime la portion de texte de la *fin* de la valeur de *variable* qui correspond au modèle.

Exemples 16.9

- (a) Si `var` contient la valeur `/usr/local/bin`,

`${var##*/}`

est égal à `bin` et

`${var#*/}`

est égal à `usr/local/bin` (sans le `/` du début)

- (b) Si on travaille avec *ksh*, on peut placer

`PS1='${PWD##$HOME/}$ ' '`

dans le fichier `.profile` pour avoir un prompt qui indique le répertoire courant sans être trop long pour les noms de répertoire qui sont sous le répertoire courant.

16.5.4 Modification interactive de la valeur d’une variable sous *zsh* (*vared*)

zsh offre une facilité pour modifier en interactif la valeur d’une variable avec la commande *vared* (“*variable editor*”). Par exemple,

`vared PATH`

permet de modifier la valeur de la variable `PATH` à la manière de *emacs*.

16.5.5 Tableaux sous *ksh* et *zsh*

Les shells *ksh* et *zsh* permettent d’utiliser des tableaux.

La syntaxe n’est pas la même pour ces deux shells. Nous ne donnerons que des exemples avec la syntaxe de *zsh* :

`samoa(~) tableau=(abc def)`

`samoa(~) echo $tableau`

`abc def`

`samoa(~) echo $tableau[1]`

`abc`

`samoa(~) echo $tableau[2]`

`def`

Quatrième partie

Programmation

Chapitre 17

Programmation des shellscrip

Ce chapitre étudie les commandes les plus souvent utilisées dans l’écriture des shellscrip

17.1 Tests divers (test, [...], [[...]])

Cette commande permet de tester une condition. Elle est souvent utilisée par les structures de contrôle du shell. Cette commande est interne ou externe selon les shells et les versions d’Unix et les options peuvent varier suivant les versions (consultez le manuel en ligne).

```
test condition
ou
[ condition ]
```

renvoie 0 si *condition* est vraie et une valeur différente de 0 si *condition* est fausse.

Dans la deuxième forme, “[” est le nom d’une commande synonyme de la commande “test”. Le “]” est là pour “faire joli”. Il ne faut pas oublier les espaces de part et d’autre de *condition*.

Quelques conditions (attention à bien séparer les différents “mots” par des espaces) :

-d <i>fichier</i>	vrai si <i>fichier</i> est un répertoire
-f <i>fichier</i>	vrai si <i>fichier</i> est un fichier ordinaire
-r <i>fichier</i>	vrai si on a l’autorisation de lire <i>fichier</i>
-w <i>fichier</i>	vrai si on a l’autorisation d’écrire dans <i>fichier</i>
-x <i>fichier</i>	vrai si on a l’autorisation d’exécution dans <i>fichier</i>
-s <i>fichier</i>	vrai si <i>fichier</i> n’est pas vide
<i>chaîne</i>	
-n <i>chaîne</i>	vrai si la chaîne n’est pas vide
-z <i>chaîne</i>	vrai si la chaîne est vide
<i>n1</i> -eq <i>n2</i>	vrai si les 2 nombres entiers <i>n1</i> et <i>n2</i> sont égaux. A la place de eq on peut avoir ne, gt, ge, lt, le (pour \neq , $>$, \geq , $<$, \leq).

chaîne1 = *chaîne2*
vrai si les 2 chaînes sont égales
chaîne1 != *chaîne2*
vrai si les 2 chaînes ne sont pas égales
-t [*numéro*]
vrai si le *numéro* est le numéro d’une voie ouverte sur un terminal (par défaut, *numéro* est égal à 1, c’est-à-dire correspond à la sortie standard)
-L *fichier*
vrai si *fichier* est un lien symbolique. Cette option existe pour la commande interne de ksh et de zsh mais pas pour toutes les commandes externes. Elle peut aussi s’appeler “-h” pour certaines commandes externes.

Remarque 17.1

Si un lien symbolique pointe sur un répertoire, l’option “-d” est vraie sur ce lien. Il faut donc commencer par tester si c’est un lien symbolique si on veut faire la différence.

On peut combiner ces conditions avec les opérateurs logiques

-a *et* logique
-o *ou* logique
! *négation* logique

On peut regrouper avec des parenthèses (entourées d’un espace de part et d’autre et précédées par \ pour qu’elles ne soient pas interprétées par le shell).

Exemples 17.1

- (a) `test -d $1 && echo OUI || echo NON`
- (b) `test \(-d fich -o -f fich2 \) -a $var = oui`

De plus *ksh* et *zsh* offrent un mécanisme interne pour tester les expressions conditionnelles :

`[[expression conditionnelle]]`
(des espaces sont nécessaires entre “*expression conditionnelle*” et les doubles crochets). Les expressions sont les mêmes que pour `test` mais le *et* logique se traduit par “&&” et le *ou* logique par “|”.

On dispose aussi de quelques ajouts. Par exemple, “<” et “>” permettent de comparer deux chaînes de caractères dans l’ordre lexicographique (ordre du dictionnaire), “*chaîne* = *modèle*” est vrai si *chaîne* correspond à *modèle* qui peut contenir les caractères spéciaux utilisés dans la génération des noms de fichiers (voir 4.5).

Aucune génération des noms de fichiers n’est effectuée par le shell dans “*expression conditionnelle*”.

Exemple 17.2

`[[$fichier = *.c]]`

17.2 Décaler les paramètres de position (shift)

`shift [n]`

décale les paramètres de position de *n* positions. Par défaut *n* est égal à 1 : \$2 devient \$1, \$3 devient \$2, etc.

shift est souvent utilisé par les structures de contrôle de répétition pour utiliser à tout rôle les différents paramètres de position (voir exemples en 17.4.4).

Exemple 17.3

Si le shellscript “decale” contient

```
shift
echo $1
```

l’exécution de “decale 1 2” affichera 2.

17.3 Sortie d’un shellscript (exit)

`exit [n]`

fait sortir du shell (donc en général du shellscript qui est exécuté par le shell ; voir 14.1.1) en renvoyant le code *n* (voir 14.1.6).

Exemple 17.4

```
exit 1
```

Si le shellscript ne se termine pas par un `exit` ou si `exit` n’est pas suivi d’un nombre, il renvoie le code de la dernière commande qu’il a exécutée.

Remarque 17.2

Il faut se méfier des constructions des shells qui créent implicitement un sous-shell (voir 14.3 ou 17.4.7). Si une commande *exit* s’exécute dans un de ces sous-shells, elle ne fera sortir que du sous-shell et pas du shellscript.

17.4 Les structures de contrôle

17.4.1 if .. then .. else .. fi

Les commandes Unix renvoient toute un code retour qui est un nombre entier (voir 14.1.1).

En général, si la commande s’est bien déroulée, elle renvoie 0 ; s’il y a eu une erreur ou un déroulement anormal de la commande, le code renvoyé est négatif ou strictement positif.

La commande `if` prend en compte ce code renvoyé par les commandes.

```

if liste-commandes ; then
    liste-commandes
else
    liste-commandes
fi

```

teste la valeur renvoyée par la liste de commande qui suit le *if* (code de la dernière commande de la liste exécutée). Si la valeur est nulle, la liste de commandes qui suit le *then* est exécutée, sinon c'est la liste qui suit le *else* qui est exécutée.

Remarque 17.3

Les mots clés *if*, *then*, *else*, *fi* doivent apparaître en début de ligne (précédés seulement d'espaces ou de tabulations) pour être reconnus par le shell. Si on veut les mettre à la suite d'autres instructions sur la même ligne il faut les faire précéder d'un ";". La présentation choisie dans ce cours suit le format couramment utilisé en langage C (*else* aligné avec le *if*). Le *then* est placé sur la même ligne que le *if* et il est donc précédé d'un ";".

Exemples 17.5

```

(a) if cp rep1/* rep2 ; then
    :
else
    echo "Impossible de copier les fichiers" >&2
fi

```

(b) Écrivons un shellscript "oui" qui pourra être utilisé par d'autres shellscripts. Il renverra 0 si l'utilisateur répond oui et il renverra 1 sinon :

```

read reponse
if test "$reponse" = oui ; then
    exit 0
else
    exit 1
fi

```

On peut utiliser ce shellscript dans un autre shellscript comme suit :

```

echo -n "Confirmez-vous la suppression ? "
if oui ; then
    rm *
fi

```

Remarque 17.4

Puisqu'en l'absence de commande *exit*, un shellscript sort avec le code retour de la dernière commande exécutée, le shellscript "oui" pourrait aussi s'écrire :

```

read reponse
test "$reponse" = oui

```

Variantes de if

```

if liste-commandes ; then
    liste-commandes
fi

```

```

if liste-commandes ; then
    liste-commandes
elif liste-commandes ; then
    liste-commandes
elif ...
...
...
else
    liste-commandes
fi

```

Cette structure est un raccourci plus simple et plus lisible pour la structure suivante :

```

if liste-commandes ; then
    liste-commandes
else
    if liste-commandes ; then
        liste-commandes
    else
        if ...
        ...
        ...
    else
        liste-commandes
    fi
fi
fi

```

Exemple 17.6

```

if test "$reponse" = oui ; then
    exit 0
elif test "$reponse" = non ; then
    exit 1
else
    exit 2
fi

```

17.4.2 case ... esac

```
case mot in
    modèle1) liste-commandes ;;
.
.
    modèlen) liste-commandes ;;
esac
```

case est une instruction de choix multiple : le premier modèle rencontré qui correspond à la valeur de *mot* indique la liste de commandes à exécuter. Si *mot* ne correspond à aucun des modèles, l'exécution se poursuit à l'instruction qui suit "esac".

Les modèles sont de la forme :

```
val1 | val2... | valp
```

Dans les modèles, on peut utiliser les caractères spéciaux "? * []" comme dans les noms de fichiers (voir 4.5). On peut ainsi tester si une valeur commence par un chiffre par le modèle "[0-9]*".

Exemple 17.7

Le shellscript "oui" donné en b peut s'écrire :

```
read reponse
case "$reponse" in
    o|O|oui|OUI) exit 0 ;;
    n|N|non|NON) exit 1 ;;
    *) exit 2 ;;
esac
```

17.4.3 for ... do ... done

Il est fréquent d'avoir un traitement qui se répète pour chaque paramètre d'une commande (par exemple, "lpr fich1 fich2"). La commande *for* permet de traiter successivement tous les paramètres d'une commande. Elle permet aussi de répéter un traitement pour plusieurs valeurs différentes.

```
for variable [in mots...]
do
    liste-commandes
done
```

variable prend à tour de rôle pour valeur chaque mot de la liste *mots* qui suit "in". Pour chaque valeur de *variable*, *liste-commandes* est exécutée.

Si "in mots..." n'est pas présent, *variable* prend pour valeur chacun des paramètres de la commande (ce qui est équivalent à : *for variable* in "\$@" ; voir 15.2.6).

Remarque 17.5

Une boucle "for" sans "in" ne modifie pas les valeurs de \$1, \$2,...

Exemples 17.8

(a)

```
for i in 1 2 3
do
    echo $i
done
```

(b) Pour voir la différence entre \$* et \$@, exécutez le shellscript suivant avec quelques paramètres, remplacez ensuite \$* par \$@ et exécutez le à nouveau avec les mêmes paramètres. Notez la différence. Dans la première version on n'a qu'un seul "tour" car "\$*" est considéré comme un mot unique.

```
for i in "$*"
do
    echo $i
done
```

(c) Le shellscript "cherche" suivant affiche les chemins d'accès des fichiers donnés en argument :

```
for fichier
do
    find / -name "$fichier" -print
done
```

Il suffira de taper

```
cherche fich1 fich2 fich3
```

pour faire afficher les chemins d'accès des fichiers spécifiés.

Si on veut ajouter quelques tests et entrer la racine de la recherche en premier paramètre, le shellscript cherche devient :

```
if test $# -lt 2 ; then
    echo "$0 : Il faut au moins 2 arguments" >&2
    exit 2
fi
if test ! -d $1 ; then
    echo "$0 : $1 n'est pas un repertoire" >&2
    exit 3
fi
racine=$1
shift
for fichier
do
    find $racine -name "$fichier" -print
done
```

- (d) Pour exécuter un traitement (ici afficher le nom) sur tous les fichiers d'un répertoire on utilise la substitution de commande :

```
for i in `ls -A`
do
    echo $i
done
```

17.4.4 while ... do ... done

```
while liste-commandes1
do
    liste-commandes2
done
```

exécute *liste-commandes2* tant que *liste-commandes1* renvoie un code 0.

Exemples 17.9

- (a)

```
while test "$1"
do
    echo $1
    shift
done
```

affiche tous les paramètres, un par ligne. Retenez le test sur \$1 pour repérer le dernier paramètre.

- (b) Si le shellscript “coucou” contient :

```
while :
do
    echo coucou
    sleep 30
done
```

Si on tape

```
coucou &
```

il s'affichera “coucou” sur l'écran toutes les 30 secondes (ne pas rediriger sur le terminal du collègue !). La commande interne “:” est une commande qui ne fait rien et renvoie le code retour 0 (voir 17.6).

- (c) Pour rechercher dans l'arborescence une liste de fichiers (\$2) situés sous des répertoires différents (\$1), on peut utiliser le shellscript suivant (le shellscript “cherche” a été écrit en c) :

```
while [ "$1" ]    # teste la fin de la liste
```

```
do
    cherche "$1" "$2"
    shift 2
done
```

- (d) Lecture ligne à ligne d'un fichier structuré en lignes :

```
while read ligne
do
    echo $ligne
done < fichier
```

17.4.5 until ... do ... done

Est semblable à while mais la sortie de la boucle s'effectue si la liste de commandes qui suit until renvoie une valeur nulle.

17.4.6 Instructions liées aux boucles (continue, break)

```
continue [n]
```

permet de sauter à la fin de la boucle et de recommencer une nouvelle boucle.

```
break [n]
```

provoque la sortie de la boucle en cours d'exécution et passe à l'instruction qui suit la boucle.

On peut ajouter un nombre entier *n* à la suite de ces 2 instructions pour indiquer que l'action porte sur des boucles externes. Par exemple, “break 2” sortira de la boucle qui englobe la boucle dans laquelle cette instruction est écrite.

17.4.7 Problèmes avec les boucles redirigées et les pipes

Lorsqu'une boucle est redirigée vers un fichier, le shell l'exécute dans un sous-shell. On est de même avec les commandes d'un pipe (sauf peut-être la dernière commande).

Ceci implique les deux problèmes suivants :

- une commande *exit* ne sortira pas du shellscript mais seulement de la boucle ou du pipe,
- les affectations de variables à l'intérieur de la boucle ne seront pas connues en dehors de la boucle ou du pipe.

Remarque 17.6

Malgré tout on n'est pas obligé d'exporter une variable pour qu'elle soit connue des sous-shells engendrés par le shell pour exécuter la boucle redirigée (vérifiez-le tout d'abord même pour la version du shell que vous utilisez).

Voici des méthodes (de type “bidouillage”) pour contourner ces deux problèmes dans le cas d’une boucle redirigée :

- Pour le problème du *exit*, il n’y a pas de solution miracle. On doit s’adapter à chaque cas particulier. On peut envoyer un code retour spécial avec la commande *exit* à l’intérieur de la boucle (par exemple 5) et on teste le code retour juste à la sortie de la boucle. On peut ainsi sortir du shellscript si le code retour lu est le code spécial choisi.
- Pour l’affectation de variable, il n’y a pas non plus de solution simple. On peut écrire l’affectation dans un fichier temporaire que l’on exécute avec la commande “.” (voir 14.11.2) à la sortie de la boucle. Par exemple (la valeur de \$\$ est la même dans les sous-processus du pipe mais il vaut mieux vérifier),

```
echo "var=valeur" > ftemp$$
```

On peut aussi dans certains cas utiliser la substitution de commande comme il est décrit ci-dessous pour le pipe. C’est alors la solution la plus “propre”.

- Pour le pipe, une solution est d’utiliser un fichier temporaire qui contient les informations que l’on veut faire passer au shell qui a appelé le pipe. Dans certains cas, une autre solution est de mettre le pipe dans un shellscript à part (ou une fonction), de faire afficher par ce shellscript les informations que l’on veut récupérer, et d’utiliser une substitution de commande pour récupérer ces valeurs. Cette dernière solution est aussi utilisable dans le cas d’une boucle redirigée.

17.5 Interception des signaux

```
trap [liste-commandes] n...
```

Le plus souvent, un processus s’interrompt quand il reçoit un signal (voir 2.2.1). La commande *trap* permet un comportement différent à la réception du signal : *liste-commandes* sera exécutée si le programme reçoit un des signaux dont le numéro est indiqué dans la liste des *n* ; ensuite, le programme continue à l’endroit où il était au moment de la réception du signal (sauf si une instruction comme *exit* a été exécutée entre temps).

Si *n* est 0, la liste de commandes sera exécutée à la sortie du shell en cours (en général, à la fin de l’exécution du fichier de commande qui contient l’instruction *trap*) ou à la sortie de la fonction (voir 16.3) dans laquelle a été défini le *trap*.

Si *liste-commandes* est la chaîne nulle (indiquée explicitement par ‘’ : 2 apostrophes accolées), le signal est ignoré.

Si *liste-commandes* est absente, le signal sera à nouveau traité normalement. “*trap*” (sans aucun paramètre) affiche la liste des commandes associées à chacun des signaux.

La liste de commandes du *trap* est lue et interprétée une première fois quand le shell lit la ligne qui comprend la commande *trap* et le résultat de la première interprétation est interprétée une deuxième fois si la liste de commande est exécutée après la réception d’un signal.

17.5. INTERCEPTION DES SIGNAUX

Ces signaux sont pour la plupart des signaux d’interruption pour cause de déroulement anormal (erreur au niveau du bus, mémoire insuffisante, etc.) ou d’interruption par l’utilisateur (l’appui sur la touche d’annulation correspond au signal numéro 2). La commande *kill* (étudiée en 7.2) permet d’envoyer un signal à un processus repéré par son numéro.

Exemples 17.10

(a)

```
trap '' 2 # les deux apostrophes designent la chaine vide
```


ignore l’interruption par la touche d’annulation

(b)

```
trap 2
```


rétablit la prise en compte de cette touche

(c)

```
temp=temp$$
trap 'rm -f $temp ; exit 2' 1 2 15
```


supprime le fichier temporaire dont le nom est composé de la chaîne “temp” suivie du numéro de processus, à la fin du programme, si l’utilisateur appuie sur la touche d’annulation ou si le programme a été interrompu par un “kill” ou par la déconnexion de l’utilisateur.

Portée d’un *trap* : Le *trap* est en activité dans un shellscript à partir du moment où il est lu et jusqu’à ce qu’un autre *trap* sur le même signal soit lu. Les sous-shell n’héritent pas d’un traitement du signal initié par un *trap*. Cependant, si un shellscript ignore un signal, le signal ne sera pas transmis à ses fils.

Exemples 17.11

(a) Lancez le programme suivant, tapez sur la touche d’annulation et voyez le message s’afficher à l’écran.

```
trap "echo $0 : fin de programme ; exit 2" 2
coucou
```

où coucou est le shellscript suivant :

```
while :
do
    echo coucou
    sleep 5
done
```

(b) Le schéma ci-dessous permet d’interrompre “commande” en tapant la touche d’annulation, sans interrompre le programme principal ; l’exécution reprend à la suite du programme.

```
trap "trap '' 2" 2 # les guillemets pour entourer la chaine
```

```

# avec les deux apostrophes
commande
trap '' 2
Suite du programme...
```

Si le premier trap était `trap '' 2`, on ne pourrait pas interrompre la commande. On pourrait aussi mettre `trap : 2` comme premier trap mais il pourrait peut-être y avoir un problème si la touche d'annulation était tapée pendant l'exécution du deuxième trap (placé après la commande) : le signal 2 ne serait plus ignoré dans la suite du programme.

17.6 Commandes internes diverses

```
[:
```

commande nulle (elle peut être suivie de commentaires sur la même ligne). Elle renvoie le code 0 ce qui permet de construire des boucles infinies (voir exemple de `while` de la page 130).

```
eval chaînes...
```

les chaînes sont interprétées par le shell et elles sont ensuite exécutées comme si elles avaient été tapées au clavier (donc avec une nouvelle interprétation du shell). Cette commande permet de construire des commandes complexes morceau par morceau et des les exécuter ensuite. Elle permet aussi des astuces qui sont liées à une double interprétation du shell comme dans l'exemple suivant.

Exemple 17.12

```
eval dernier='$'$#
```

affecte à la variable “dernier” la valeur du dernier paramètre de position du shellscript. En effet, quand le shell lit la commande et l'interprète, la commande devient

```
eval dernier=$3
```

(s'il y avait par exemple 3 paramètres de position) car le shell effectue la substitution de variable (`$#`).

La commande interne “eval” est ensuite lancée : elle commence par interpréter la chaîne “`dernier=$3`” comme le ferait le shell. Cette chaîne devient “`dernier=toto`” (si “toto” est la valeur du troisième paramètre).

L'affectation est ensuite exécutée et on obtient bien ce que l'on voulait.

```
exec commande
```

la commande est exécutée à la place du shell sans création de nouveau shell.

On a vu en 16.4.2 que `exec` permet aussi de faire des redirections définitives dans le shell.

```
wait [n]
```

attend le processus fils de pid `n`. Si `n` n'est pas spécifié, attend tous les processus fils.

Cette commande permet de lancer des processus en arrière-plan et d'en attendre certains avant de continuer une action. Il peut être utile de conserver le pid d'un processus lancé en arrière-plan dans une variable à l'aide de la variable spéciale `!` (voir 15.2.6).

17.7 Récursivité dans l'exécution des shellscripts

Un shellscript (ou une fonction) peut s'appeler lui-même. Par exemple le shellscript “`dirr`” suivant affiche tous les sous-répertoires de l'arborescence d'un répertoire donné en paramètre en décalant les différents niveaux de sous-répertoires :

```

dir=${1-.}
indent=$2
for fich in `ls -A $dir`
do
    if [ -d $dir/$fich ]; then
        echo "$indent"$fich
        $0 $dir/$fich "$indent"
    fi
done
```

Remarques 17.7

- (a) L'utilisation de `$0` au lieu du nom du shellscript assure que le shellscript fonctionnera même si on change son nom.
- (b) L'exécution de ce shellscript risque de provoquer une erreur si l'arborescence du répertoire est profonde et si la mémoire allouée à la pile est trop faible.

17.8 Calculs, traitement des chaînes de caractères

`ksh` et `zsh` fournissent la commande interne `let` pour effectuer des calculs arithmétiques. Cette commande n'existe pas dans `sh` mais on peut utiliser la commande `expr`. `expr` permet aussi d'extraire des sous-chaînes d'une chaîne de caractères.

17.8.1 Commande expr

```
expr arguments...
```

évalue les arguments comme une expression. Le résultat est envoyé sur la sortie standard. Ce n'est pas une commande interne au shell mais elle est très utile dans l'écriture des shellscripts.

Attention, sauf pour les options ci-dessous où le code retour est indiqué, le code retour renvoyé est 0 lorsque la valeur envoyée sur la sortie standard est différente de 0 et 1 si la valeur est 0.

Les calculs se font sur des entiers codés sur 32 bits en complément à 2.

arguments est une expression comprenant des opérateurs. On peut utiliser les opérateurs suivants. Il ne faut pas oublier de les protéger contre une interprétation du shell s'ils sont des caractères spéciaux pour le shell (par exemple, les caractères `< > * & |`).

Voici ces opérateurs par ordre de priorité croissante, regroupés par groupes d'égales priorités :

exp1 \ | exp2

si *exp1* est non nulle et non vide, elle est envoyée sur la sortie standard, sinon si *exp2* est non nulle et non vide, elle est envoyée sur la sortie standard, sinon 0 est envoyée sur la sortie standard.

exp1 \& exp2

si *exp1* et *exp2* sont non nulles et non vides, *exp1* est envoyée sur la sortie standard, sinon 0 est envoyée sur la sortie standard.

exp1 op_comp exp2

où *op_comp* est un opérateur de comparaison :

`< <= = != >= >`

La comparaison est numérique si les deux expressions sont numériques ; elle est lexicographique sinon. Envoie 1 sur la sortie standard si l'expression est vrai et 0 sinon.

exp1 + exp2

exp1 - exp2 addition et soustraction

exp1 * exp2

exp1 / exp2

exp1 \% exp2 multiplication, division et modulo

Autres *arguments* :

chaîne (une seule chaîne de caractère) envoie *chaîne* sur la sortie standard et le code retour 0 si *chaîne* est non vide ; envoie une ligne vide et le code retour 1 si *chaîne* est vide.

chaîne : exp-reg

renvoie le nombre de caractères de *chaîne* qui peuvent être désignés par l'expression régulière *exp-reg* (voir chapitre 11). *exp-reg* doit représenter le début de *chaîne* et il ne faut donc pas commencer *exp-reg* par le caractère `^`.

Si une partie de l'expression régulière est parenthésée par `(` et `)`, *exp-reg* renvoie la sous-chaîne de *chaîne* qui correspond à cette partie. Le code retour 1 est renvoyé si l'expression régulière *exp-reg* ne correspond à aucune sous-chaîne de *chaîne* et 0 sinon.

On peut regrouper les expressions en les parenthésant par `(` et `)`.

Exemples 17.13

- (a) `a='expr $a + 1'`
incrémente a de 1.

17.8. CALCULS, TRAITEMENT DES CHAÎNES DE CARACTÈRES

- (b) Le programme suivant affiche les nombres entiers de 1 à 9 :

```
a=1
while [ $a -lt 10 ]
do
    echo $a
    a='expr $a + 1'
done
```

- (c) `expr "$a" : ".*"`
affiche le nombre de caractères de la chaîne contenue dans la variable a.

- (d) `expr 123abc : '[0-9]*'`
affiche 3.

- (e) Le shellscript “option” suivant affiche le nombre n associé à une option “-n”. Il renvoie le code retour 1 si le paramètre ne commence pas par le caractère “-” :

```
expr $1 : '-\(.*)'
```

17.8.2 Commande let

Cette commande n'existe pas en *sh*. Elle est disponible dans *ksh* et *zsh*.

`let expression arithmétique`

permet d'affecter des valeurs numériques à une variable.

L'*expression arithmétique* doit être un seul mot pour le shell (on peut l'entourer de guillemets pour cela). Il ne faut pas faire précéder les noms des variables du shell par le signe \$ pour désigner leur contenu.

Exemples 17.14

- (a) `let a=10+b`

- (b) `let "a = 10 * a"`

`((expression arithmétique))`

est équivalent à

`let "expression arithmétique"`
(voir 14.8)

Exemples 17.15

- (a) `((a=10+b))`

- (b) `((a = 10 * a))`

Pour des calculs plus complexes on utilisera la commande *bc*.

17.9 Traitement des chaînes de caractères

Le langage *Perl* permet d'effectuer de nombreuses manipulations sur les chaînes de caractères. Le langage est complexe et n'est pas étudié dans ce cours. Son usage doit être envisagé pour les programmes qui comportent beaucoup de traitements de chaînes de caractères. Il est en particulier très utilisé dans les programmes "CGI" (*Common Gateway Interface*) installés sur les serveurs Web (voir cours photocopiés sur les réseaux).

La commande habituelle utilisée pour la manipulation de chaînes de caractères est la commande *expr* (voir 17.8.1).

Voici des méthodes (plus ou moins) simples pour extraire une sous-chaîne de caractères. Pour ces exemples, on extrait l'heure les minutes et les secondes en utilisant la commande *date* et la substitution de commande (voir 14.7) :

```
heure='date | cut -c12-19'
heure='date | awk '{print $4}'
heure='date | read a b c d e; echo $d'
heure='date | sed 's/[~]* [~]* [~]* \([~]*\).*\/1/'
heure='set `date` ; echo $4'
heure='expr "\" : ' [~]* [~]* [~]* \([~]*\)''
```

On peut aussi utiliser la variable IFS. Voici des lignes de programme pour afficher le nom de la machine d'une adresse internet :

```
nom=samoa.unice.fr
IFS=.
set $nom
echo $1
```

17.10 Aide pour traiter les options (getopts)

La commande *getopts* est une commande interne qui aide à récupérer les valeurs des paramètres de position des shellscripts dont la syntaxe suit la syntaxe habituelle des commandes Unix.

On sait que les commandes Unix acceptent plusieurs formats différents. On peut écrire par exemple "ls -la" ou "ls -l -a". Quand une option nécessite un complément, ce complément peut être accolé à l'option ou séparé de l'option par un espace ("-P1rv" ou "-P 1rv" par exemple). La récupération des options d'un shellscript nécessite donc un traitement assez complexe. La commande *getopts* facilite grandement la tâche du programmeur.

L'ancienne commande *getopt* effectuait le même type de traitement mais ne fonctionnait pas correctement dès qu'un des arguments était composé de plusieurs mots (comme le complément de l'option -s dans "mail -s "Sujet du message" toto").

```
getopts options nom-variable [paramètres...]
```

getopts analyse les paramètres de position (ou la chaîne constituée de *paramètres...* si elle est donnée). *getopts* traite une à une toutes les options, à chaque fois qu'on le lance. Il

met dans la variable du shell *nom-variable* la prochaine option et le numéro de prochain argument à traiter dans la variable OPTIND.

La première fois que *getopts* est appelée, OPTIND vaut 1 et ensuite OPTIND est incrémenté à chaque fois que l'on appelle *getopts*, au fur et à mesure du traitement des différents paramètres. Si une des options nécessite un argument (comme l'option -P évoquée ci-dessus), cet argument est rangé dans la variable OPTARG.

Si une option illégale est rencontrée, *nom-variable* reçoit la valeur "?".

Quand toutes les options ont été passées en revue, *getopts* renvoie un code retour non nul. L'option spéciale "--" peut indiquer la fin des options (on peut ainsi indiquer un argument qui n'est pas une option et qui commence par "--").

Les options acceptées sont indiquées dans la chaîne *options*. Le caractère "." est accolé derrière les options qui doivent être suivies d'un complément.

Voici un exemple d'utilisation de *getopt* en début de shellscript. Ce shellscript est supposé accepter les options simples -a et -b et les options -s et -f avec des compléments. On notera que l'on analyse d'abord tous les paramètres avant de commencer les traitements. On n'est ainsi pas tributaire de l'ordre dans lequel l'utilisateur a donné les paramètres sur la ligne de commande.

```
opta=0; optb=0; optf=0; opts=0
fichier= # complement pour option -f
sujet=   # complement pour option -s
while getopts abs:f: argument
do
    case $argument in
        a) opta=1 ;;
        b) ooptb=1 ;;
        f) optf=1
           fichier="$OPTARG";;
        s) opts=1
           sujet="$OPTARG";;
        \?) echo "'basename $0': Usage: \
               'basename $0' [-ab] [-s sujet] -f fichier-adresses [message]" >
           exit 2;;
    esac
done
shift `expr $OPTIND - 1`

if [ $optf -eq 0 ] ; then
    ...
```

17.11 Mise au point des shellscripts (set -xv)

```
set -v
```

fait afficher les lignes du shellscript au moment où elles sont lues par le shell. **set +v** annule cette option.

```
set -x
```

fait afficher les commandes (et leurs arguments) au moment de leur exécution, *après* l'interprétation du shell. **set +x** annule cette option.

Ces commandes peuvent être insérées de part et d'autre des lignes que l'on veut "suivre à la trace" dans les shellscripts que l'on veut mettre au point.

Si on veut suivre à la trace l'exécution entière d'un shellscript, le plus simple (car on évite de modifier le shellscript) est de lancer le shellscript par (on remplacera *sh* par *zsh* si le shellscript doit être exécuté par *zsh*).

```
sh -x nom-shellscript
ou
sh -v nom-shellscript
```

Index

#, 32	.zshenv, 111
#!, 101	.zshrc, 111
\$, 58	:, 118, 134
\$!, 108	;, 113
\$#, 108	<, 94
\$\$, 108	<&n, 117
\$(...), 98	<<, 115
\$*, 108	=, 105
\$?, 108	>, 93
\$@, 108	>!, 94
\$n, 104	>&n, 117
\$variable, 105	>>, 94
\${#variable}, 119	?, 31, 59
\${=variable}, 106	[, 31, 59
\${variable##modèle}, 119	[(test), 123
\${variable#modèle}, 119	[[...]], 124
\${variable+mot}, 118	\, 31, 58, 99
\${variable-mot}, 118	, 99
\${variable=mot}, 118	\(...\), 60
\${variable?mot}, 118	%n, 97
\${variable%%modèle}, 120	\n, 60
\${variable%modèle}, 119	\{m, \}, 60
\${variable}, 106	\{m,n\}, 60
&, 96, 113	\{m\}, 60
&&, 113	~, 31
', 99	
((...)), 137	
(...), 95	
*, 30, 59	
+, 59	
., 58, 102	
.plan, 38	
.project, 38	
.zlogin, 111	
.zprofile, 111	
	a2ps, 48
	administrateur, 11
	affectation, 105
	afficher, 108

afficher le contenu d'un fichier, 44
 afficher les octets d'un fichier, 45
 afficher page à page, 45
alias, 97
 appels système, 5
 arrière-plan, 13, 96
at, 43, 96
 autorisations, 35, 36
 autorisations d'un fichier, 15
awk, 81

batch, 96
bc, 137
bg, 97
 boucles redirigées, 131
 Bourne shell, 6
break, 131

C-shell, 6
 calculs, 135
 calculs mathématiques, 137
cancel, 47
 caractères spéciaux pour le shell, 30
case, 128
cat, 44
cd, 30
CDPATH, 107
 CGI, 138
 chaîne de caractères, 135, 138
chmod, 35
cmp, 78
 code retour, 78, 93, 125
comm, 88
 commande externe, 6
 commande interne, 6, 92, 134
 commentaire, 32
 comparer deux fichiers, 78
 compatibilité de zsh, 92
 complétion des commandes, 24
compress, 54
 compter les caractères, mots et lignes, 74
 concaténer plusieurs fichiers, 44
 configurer zsh, 92

continue, 131
 convertir des caractères, 75
 copier des fichiers, 50
 copier l'arborescence d'un répertoire, 56
 core, 13, 34
cp, 50, 56
cpio, 57
 créer un répertoire, 55
csch, 6
csplit, 88
cut, 76

daemon, 13
date, 37
 début d'un fichier, 79
 décaler les paramètres, 125
 demand paging, 12
 démon, 13
 dépersonnaliser, 99
 déplacer des fichiers, 53
 descripteur de fichier, 116
df, 39
diff, 78
 différences entre deux fichiers, 78
DISPLAY, 112
 disquette, 53
 driver, 5, 29
 droits sur un fichier, 16
du, 39

echo, 108
ed, 115
 éditeur de texte, 62
eject, 53
emacs, 62
EMACSLOADPATH, 63
 entrée standard, 93
 environnement, 11, 92, 104, 109, 110, 119
 Envoyer un fichier, 48
 erreur standard, 93
eval, 134
 exécution d'un shellscrip, 100
exec, 116, 134

exit, 19, 125, 131
export, 109
expr, 135
 expression conditionnelle, 124
 expression régulière, 58
 extraire une sous-chaîne, 135

false, 113
fg, 97
 fichier
 type de, 7, 29
 fichier inclus, 115
 fichier ordinaire, 7
 fichier spécial, 8, 29
file, 32
 fin d'un fichier, 79
find, 32
finger, 38
fold, 88
 fonction, 114
for, 128

génération des noms de fichiers, 30
 gestion des jobs, 97
getopt, 138
getopts, 138
grep, 74
 groupe d'un fichier, 15
 groupe d'un processus, 13
 groupe d'utilisateurs, 9
gzip, 54

head, 79
 historique de Unix, 3
HOME, 30, 106
hostname, 37

i-node, 8, 14, 51
 identificateur d'une variable, 105
if, 125
IFS, 107
 imprimantes disponibles, 46
 imprimer, 45
 inhiber l'interprétation du shell, 99

interprétation du shell, 99
 interpréteur de commandes, 5

jobs, 97
join, 88

kill, 22, 42
 Korn-shell, 6
ksh, 6

lancer une commande, 22
last, 38
 lecteur de disquettes, 53
 lecture ligne à ligne d'un fichier, 131
let, 137
 lien, 30, 52
 lien "hard", 51
 lien symbolique, 52
 lien symbolique pour un répertoire, 56
 lignes consécutives distinctes, 79
 lignes consécutives identiques, 79
 liste de commandes, 113
ln, 52, 56
locate, 34
lpq, 47
lpr, 46
lprm, 47
lpstat, 46
ls, 27

mail, 48
 majeur, 29
man, 21
mcopy, 53
mdir, 53
 message d'erreur, 94
 mineur, 29
 mise au point, 139
 mise en page, 48
mkdir, 55
 mode "dired" d'emacs, 68
 mode d'accès à un fichier, 15
 modifier la valeur d'une variable, 120
 monter un système de fichiers, 6

more, 45
 mot de passe, 18
mv, 53, 55

 NFS, 7
 NIS, 18, 38
 NOCLOBBER, 94
nohup, 96
 nom absolu, 22
 nom d'un fichier, 22
 nom d'une commande, 22
 nom de l'ordinateur, 37
 nom relatif, 22
 noyau, 5
NULLCMD, 118
 numéro de job, 96
 numéro de processus, 11

od, 45
 options
 traitement, 138
 options des shells, 92, 110
 ordre de priorité, 114
 OSF, 4

 paramètre, 104
 partition d'un disque, 6
passwd, 18
paste, 88
PATH, 22, 100, 107
 PC (compatible), 53
 Perl, 82
 pid, 11, 96
 pilote, 29
 pipe, 95
 pipe nommé, 8
 place libre d'un système de fichiers, 39
 place occupée par un répertoire, 39
 pliage de répertoires, 57
 Postscript, 48
pr, 48
 primitive, 5
print, 108
 \subitem printcap, 46

printenv, 110
PRINTER, 46
 processus, 11
 image, 11
 processus en cours d'exécution, 41
 programme de démarrage, 10
 prompt, 92
 propriétaire d'un fichier, 15
 propriétaire effectif, 13
 propriétaire réel, 13
 protection des fichiers, 15
ps, 41
PS1, 107, 120
PS2, 107
 pseudo-fichier, 95
pwd, 30

 racine, 22
read, 108
 recherche d'une chaîne, 74
 recherche d'une commande, 100
 recherche de fichiers, 32
 rechercher le manuel en ligne, 23
 rechercher une commande, 23
 récursivité, 135
 redirection, 44, 93, 115, 116
 redirection des erreurs, 94
 redirection sur fichier inclus, 115
 regrouper des commandes, 95
 renommer des fichiers, 53
 renommer un répertoire, 55
 répertoire
 structure interne, 14
 répertoire courant, 12
 répertoire HOME, 10
 requête d'impression, 46
return, 115
rm, 52, 55
rmdir, 55

 sauvegarder sur disquette, 43
sed, 80
set, 91, 104, 110, 139

set group id, 16
 set user id, 16
setopt, 110
 sgid, 16
sh, 6
 shell, 5, 91
 shell d'exécution d'un shellscript, 101
 shellscript, 6, 91, 100
shift, 125
 signal, 12, 42, 132
sleep, 130
 socket, 8
 Solaris, 4
sort, 73
 sortie standard, 93
split, 88
 spool, 46
 substitution de commande, 98
 suid, 16
 super-utilisateur, 11
 supprimer des caractères, 75
 supprimer des fichiers, 52
 supprimer des répertoires, 55
 supprimer des requêtes d'impression, 47
 supprimer un processus, 22, 42
 supprimer une variable, 106
 suspendre un processus, 97
 swap, 12
 système d'exploitation, 3
 nom du, 37
 système d'impression, 46
 système de fichiers, 6

tail, 79
tar, 57
TERM, 107, 112
 terminal de contrôle, 13
test, 123
 thread, 11
 touche spéciale, 19
tr, 75
 traiter les options, 138
trap, 132

tri, 73
true, 113
tty, 39
 type d'un fichier, 32
 type de fichier, 29

 uid, 9
umask, 36
unalias, 97
uname, 37
uniq, 79
 Unix BSD, 4
 Unix OSF, 4
 Unix Système V, 4
unset, 106
until, 131
uuencode, 48

 valeur par défaut d'une variable, 118
vared, 120
 variable, 118
 décomposition en mots sous zsh, 100
 valeur par défaut, 118
 variable de position, 104
 variable du shell, 105
 variable spéciale du shell, 106

wait, 134
wc, 74
whence, 23
whereis, 23
while, 130
who, 37
whoami, 37
write, 113

xemacs, 62
xterm, 103

 Yellow Pages, 38
ypcat, 38
yppasswd, 18

 Z shell, 6
zip, 54

zone de swap, 12

zsh, 6, 92