UNIVERSITÉ DE FRANCHE-COMTÉ

MODULE INFOSR

B.Herrmann et G.Laville

Système et Réseau

LICENCE INFORMATIQUE À DISTANCE

Année 2014-2015

télé-enseignement

CENTRE DE TÉLÉ ENSEIGNEMENT

Filière Informatique

Domaine Universitaire de la Bouloie

25030 Besançon Cedex (France)

Table des matières

Liste des illustrations viii

I Cours 2

1 Préliminaires 3

1.1 Documents autorisés aux examens . . . 3

1.2 Les exercices . . . 3

2 Introduction - BH - 5

2.1 Les systèmes d'exploitation . . . 5

2.1.1 Qu'est ce qu'un système d'exploitation ? . . . 5

2.1.2 Les tâches du système d'exploitation . . . 6

2.1.3 Principes des systèmes d'exploitation . . . 7

2.2 Le système UNIX . . . 9

2.2.1 Historique . . . 9

2.2.2 Caractéristiques principales . . . 9

2.2.3 Les différents UNIX ou "unices" . . . 10

2.2.4 De quoi est constitué un UNIX ? . . . 10

2.2.5 Linux . . . 10

2.3 Premiers pas avec le système UNIX . . . 11

2.3.1 Se connecter . . . 11

2.3.2 La ligne de commande . . . 13

2.4 La documentation en ligne . . . 15

2.5 L'éditeur de texte Vi . . . 15

2.6 Exercices . . . 19

3 Gestion des fichiers -GL- 21

3.1 Unix et le stockage des données . . . 21

3.1.1 Fonctionnement d'un disque dur . . . 21
3.1.2 Formatage d'un disque dur . . . 22
3.1.3 Partitionnement d'un disque dur . . . 22
3.1.4 Système de gestion de fichiers UNIX . . . 23
3.1.5 Systèmes de fichiers . . . 24
3.1.6 Organisation des fichiers sous Unix . . . 26
3.1.7 Structure arborescente standard UNIX . . . 29
3.1.8 Notion de montage . . . 29
3.1.9 Les noms de fichiers . . . 31
3.1.10 Les différents types de fichiers . . . 32
3.2 Commandes relatives aux fichiers . . . 35
3.2.1 Déplacement dans l'arborescence . . . 36
3.2.2 Création d'un nouveau fichier . . . 37
3.2.3 Visualisation d'un fichier . . . 37
3.2.4 Suppression d'un fichier . . . 39
3.2.5 Copie d'un fichier ou d'un répertoire . . . 39
3.2.6 Création d'un lien vers un fichier . . . 39
3.2.7 Déplacement d'un fichier . . . 40
3.2.8 Création et suppression de répertoires . . . 40
3.2.9 Lister le contenu d'un répertoire . . . 41
3.2.10 Concaténer le contenu de plusieurs fichiers . . . 42
3.2.11 Comparaison de fichiers . . . 42
3.2.12 Division en plusieurs parties d'un fichier . . . 43
3.2.13 Manipulation des chemins de fichiers . . . 44
3.2.14 Récupération d'informations sur des fichiers . . . 45
3.3 Droits d'accès à un fichier . . . 45
3.3.1 Les permissions . . . 45
3.3.2 Modification du propriétaire d'un fichier . . . 47
3.3.3 Modifications des permissions . . . 47
3.3.4 Les permissions par défaut . . . 49
3.3.5 Les permissions spéciales . . . 51
3.4 Exercices . . . 51
3.5 Correction des exercices . . . 52

4 Le shell et les environnements - BH - 55

4.1 Les interpréteurs de commandes ou shell . . . 55
4.1.1 Les différents shells : . . . 55
4.1.2 La soumission de commandes . . . 56
4.1.3 Les différents types de commandes . . . 57
4.1.4 La redirection des entrées/sorties . . . 58
4.2 Les fichiers de commandes . . . 61
4.2.1 Exécution de fichiers de commandes . . . 61
4.2.2 Les commentaires . . . 62
4.2.3 La mise au point . . . 63
4.2.4 Sortie de fichier de commandes . . . 63
4.2.5 Deux commandes utiles . . . 63
4.3 Les structures de contrôle . . . 64
4.3.1 Les commandes echo et read . . . 64
4.3.2 La commande if . . . 65
4.3.3 La commande test . . . 66
4.3.4 La commande Case . . . 68
4.3.5 La commande For . . . 69
4.3.6 La commande While . . . 70
4.3.7 La redirection des entrés-sorties d'une structure de contrôle . . . 70
4.4 Les variables . . . 71
4.4.1 Affectation . . . 71
4.4.2 Contenu . . . 71
4.4.3 Concaténation de variables . . . 72
4.4.4 Affectation interactive . . . 72
4.4.5 Expressions arithmétiques . . . 73
4.4.6 Portée des variables . . . 74
4.4.7 Les variables d'environnement . . . 74
4.4.8 Les variables spéciales du shell . . . 76
4.4.9 Les paramètres des procédures shell ou paramètres positionnels : . . . 77
4.4.10 Les informations des procédures shell ou variables automatiques 79
4.4.11 Substitution de commandes . . . 80
4.4.12 Les délimiteurs de chaîne . . . 82

4.4.13 Les tableaux . . . 83
4.5 Les filtres, les pipes et la manipulation de fichiers . . . 85
4.5.1 Les filtres . . . 85
4.5.2 Les expressions régulières et la commande grep . . . 89
4.5.3 Les pipes ou tubes . . . 92
4.5.4 Manipulations de fichiers . . . 93
4.5.5 La commande find . . . 96
4.6 Exercices . . . 98
4.7 Correction des exercices . . . 101

5 Gestion des utilisateurs -GL- 106
5.1 La gestion multi utilisateurs . . . 106
5.1.1 L'utilisateur . . . 106
5.1.2 Le groupe . . . 107
5.2 Environnements . . . 108
5.2.1 Environnement d'accueil de l'utilisateur . . . 108
5.3 La visualisation des utilisateurs . . . 111
5.3.1 La commande who . . . 111
5.3.2 La commande finger . . . 111
5.3.3 Environnement des utilisateurs . . . 112
5.4 Préserver les données utilisateurs . . . 113
5.4.1 Les commandes de sauvegarde . . . 113
5.5 Exercices . . . 115
5.6 Correction des exercices . . . 116

6 Les processus - BH - 118
6.1 La gestion des processus par le système . . . 118
6.1.1 Qu'est ce qu'un processus ? . . . 118
6.1.2 Création de processus . . . 119
6.1.3 Les différents types de processus . . . 120
6.1.4 Les caractéristiques d'un processus . . . 121
6.1.5 L'espace mémoire d'un processus . . . 122
6.1.6 Les processus légers . . . 124
6.2 Le droit d'exécution des processus . . . 124

6.2.1 Les permissions standards . . . 124
6.2.2 La substitution d'identité . . . 124
6.2.3 Le sticky bit . . . 126
6.2.4 En résumé . . . 126
6.3 L'ordonnancement des processus . . . 127
6.3.1 Les priorités . . . 127
6.3.2 Les différents états d'un processus . . . 128
6.4 Les signaux . . . 129
6.4.1 La gestion des signaux . . . 129
6.4.2 La commande `kill` . . . 129
6.5 La visualisation des processus . . . 130
6.5.1 La commande ps . . . 130
6.5.2 La commande ps -ef . . . 131
6.5.3 La commande ps -aux . . . 131
6.6 L'exécution des processus . . . 132
6.6.1 L'exécution asynchrone . . . 132
6.6.2 L'exécution différée . . . 134
6.7 Exercices . . . 136
6.8 Correction des exercices . . . 139

7 Réseau -GL- 143
7.1 Introduction . . . 143
7.2 Les protocoles réseau . . . 143
7.2.1 OSI . . . 144
7.2.2 TCP/IP . . . 145
7.2.3 Les principaux protocoles utilisés dans les couches TCP/IP . . . 147
7.2.4 Les services TCP/IP . . . 152
7.3 L'adressage . . . 153
7.3.1 L'adresse IP . . . 153
7.3.2 Les masques . . . 155
7.4 Les fichiers de configuration de TCP/IP . . . 160
7.4.1 Les fichiers de configuration TCP/IP . . . 160
7.4.2 Les fichiers d'autorisation . . . 161
7.5 Exercices . . . 162

7.6 Correction des exercices 164

8 L'administration et la programmation système - BH - 169

8.1 Administration système 169

8.1.1 La gestion de l'espace disque 169

8.1.2 Démarrage d'un système Unix 172

8.2 Programmation système 175

8.2.1 Passage de paramètres dans la fonction main() 175

8.2.2 Accès aux fichiers Unix 176

8.2.3 Création de processus 179

8.3 Exercices 181

8.4 Correction Exercices 182

II Annexes 187

9 Liste des principales commandes 188

10 Interprétation des caractères spéciaux 191

11 Le langage C 192

11.1 Introduction 192

11.2 Structure d'un programme 192

11.2.1 Définition et déclaration des variables 193

11.2.2 Blocs 193

11.2.3 Déclaration des fonctions 193

11.2.4 Appel de fonctions 194

11.3 Types de données 194

11.3.1 Types prédéfinis 194

11.3.2 Types composés 195

11.4 Déclaration de variables 197

11.4.1 Variables locales 197

11.4.2 Variables globales 197

11.4.3 Initialisation 197

11.4.4 Variables externes 198

11.5 Entrée/Sorties 198

11.5.1 Affichage . . . 198
11.5.2 Saisie . . . 199
11.6 Opérateurs . . . 199
11.6.1 L'opérateur d'affectation : . . . 199
11.6.2 Les opérateurs arithmétiques : . . . 199
11.6.3 Les opérateurs relationnels et logiques : . . . 200
11.6.4 Les opérateurs d'incrémentation : . . . 200
11.6.5 L'opérateur adresse : . . . 201
11.6.6 L'opérateur contenu d'adresse : . . . 201
11.6.7 Les opérateurs binaires . . . 201
11.7 Structures de contrôle . . . 202
11.7.1 If . . . 202
11.7.2 For . . . 202
11.7.3 While . . . 202
11.7.4 Répéter . . . 203
11.7.5 Cas . . . 203
11.7.6 Les ruptures de séquences . . . 204
11.7.7 Conditions . . . 204
11.8 Pointeurs . . . 205
11.9 Les chaînes de caractères . . . 206
11.10 Passage de paramètres dans les fonctions . . . 207
11.11 Conversion de type ou casting . . . 207
11.12 Compilation de programmes C . . . 209
11.12.1 Compilation classique . . . 209
11.12.2 Compilation assembleur . . . 209
11.12.3 Différentes phases d'une compilation . . . 209
11.12.4 Programmes composés de plusieurs fichiers . . . 209
11.13 Notion de gestion de projets . . . 210
11.13.1 Fichiers d'entête ou header (.h) . . . 210
11.13.2 Directives du précompilateur . . . 211
11.13.3 L'utilitaire *make* . . . 212

Bibliographie-Netographie **214**

Liste des illustrations

2.1 Représentation d'un système informatique 6

3.1 Structure d'un disque dur 22
3.2 Structure arborescente 27
3.3 Représentation physique du système de fichier illustrée par un exemple 28
3.4 Les permissions sur les fichiers 46

4.1 Illustration de la commande tee 96

6.1 Création d'un nouveau shell 120
6.2 Espace mémoire d'un processus 123
6.3 Les différents états d'un processus 128

7.1 Modèles OSI et TCP/IP confrontés 144
7.2 Le modèle 4 couches Dod –> TCP/IP 146
7.3 Structure de l'entête IP 147
7.4 Structure d'un en-tête TCP 149
7.5 Structure d'un envoi UDP 150
7.6 Communication entre 2 ordinateurs 157
7.7 Schéma du découpage réseau obtenu 168

Première partie

Cours

Chapitre 1

Préliminaires

Ce cours se veut généraliste quant à la connaissance du système d'exploitation UNIX. Ce n'est donc pas un cours LINUX, même si les supports proposés pour l'illustrer se trouvent être des Linux. Ceci est simplement dû au caractère logiciel libre des supports utilisés.

Ce cours vise à vous familiariser avec l'utilisation d'un système d'exploitation tel que UNIX mais aussi à vous faire découvrir comment fonctionne ce système en abordant les structures de données qu'il utilise ainsi que quelques uns de ces algorithmes.

Claire Grossin (PAST au laboratoire d'informatique de Besançon) a participé à l'élaboration de ce cours.

1.1 Documents autorisés aux examens

Vous trouverez en ***Annexes***, la liste des principales commandes Unix et un résumé des différentes interprétations des caractères spéciaux. **Vous disposerez de ces seuls documents lors des examens**.

1.2 Les exercices

Des exercices se trouvent à la fin de chaque chapitre. Tout au long du cours vous trouverez des **exemples** qui illustrent les notions présentées, ces exemples sont eux aussi de véritables exercices. Nous vous recommandons fortement de tester tous ces exemples.

Pour faire les exercices du cours, vous avez les solutions suivantes si vous ne possédez par un UNIX sur votre machine :
- solution 1 : utiliser la machine virtuelle à télécharger
- solution 2 : installer un Linux

La *première solution* nécessite environ 2Go de libre sur votre disque dur et d'avoir une connexion ADSL afin de télécharger la machine virtuelle (environ 200 Mo). Un

document explicatif est mis à disposition sur le site du module à la rubrique "Support pour les exercices" pour vous guidez dans cette installation. Cette solution ne vous oblige pas à installer un Unix, vous ne prenez aucun risque pour votre machine.

La *seconde solution* est pour les plus audacieux ou déjà débrouillés : il faut à ce moment là bien maîtriser la gestion de l'espace disque et disposer d'espace suffisant. Vous installez l'UNIX que vous souhaitez, un Linux sans doute, cela n'a pas d'importance pour la suite du cours qui reste généraliste. **Nous attirons votre attention sur le fait que nous n'assurons aucune aide sur le processus d'installation dans le cadre de ce cours, et que cette manipulation est faite à vos risques et périls.** Vous pouvez bien sûr échanger sur vos difficultés si vous vous lancez dans cette aventure mais **pas sur le forum du cours (toute discussion à ce sujet sera supprimée)**, ceci afin de ne pas perturber ceux qui n'auront pas choisi cette option.

Dans tous les cas, ne vous laissez pas déborder par une difficulté sur l'outil utilisé qui reste une illustration. Pour la première solution, contactez nous, pour la seconde, rabattez vous sur la la première ! ! vous ferez vos expériences une fois le cours terminé : l'objectif est avant tout de comprendre le cours, pas de maîtriser tel ou tel outil.

Chapitre 2

Introduction - BH -

Nous commençons ce chapitre par une présentation générale des systèmes d'exploitation. Le système UNIX est ensuite abordé de façon générale. La troisième section vous permet de tester les toutes premières commandes UNIX. En dernière section, l'éditeur de commandes `vi`, présent sur tous les UNIX, est présenté avec ses principales commandes.

2.1 Les systèmes d'exploitation

2.1.1 Qu'est ce qu'un système d'exploitation ?

Un système d'exploitation (Operating System en anglais, "OS" en abrégé) est une "couche" logicielle qui permet de gérer et de faire fonctionner les divers matériels qui composent l'ordinateur (processeur, mémoire centrale, horloges, terminaux, mémoires secondaires(disque, disquettes), périphériques (réseau, souris, etc).

Le système d'exploitation sert d'interface entre les applications au sens large du terme et le matériel.

On repésente un système informatique sous forme de couches, représenté sur la figure 2.1 :

1. Le **matériel** se trouve au niveau le plus bas et se compose de deux couches ou plus :
   - La couche la plus basse contient les circuits physiques.
   - Vient ensuite un logiciel de base qui contrôle ces différents dispositifs et fournit une interface plus simple à la couche suivante. Il s'agit d'un microprogramme ou interpréteur qui recherche des instructions en langage machine comme *ADD ou MOVE* et les exécute l'une après l'autre. Par exemple, pour exécuter *ADD*, il cherche ou se trouvent les nombres à additionner, les charge, les additionne et sauvegarde le résultat quelque part.
   - L'ensemble des instructions interprétées par le micro programme est appelé langage machine. Il ne fait pas réellement partie du matériel, mais est souvent considéré comme tel. Il comprend entre 50 et 300 instructions.

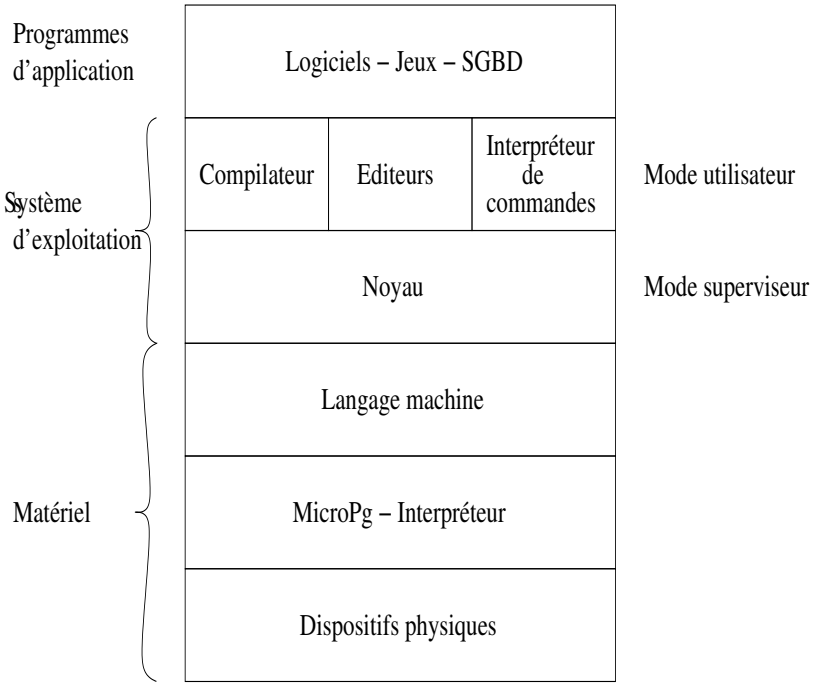

FIGURE 2.1 – Représentation d'un système informatique

2. Le **système d'exploitation** est constitué :
   - du **noyau** du système d'exploitation : qui masque toute la complexité de la programmation en langage machine proposant au programmeur un ensemble d'instructions plus simples à utiliser. Par exemple, c'est plus simple d'écrire *lire un bloc de fichier* plutôt que devoir déplacer les têtes de lecture, attendre qu'elles se positionnent, etc.
   - d'un **ensemble de logiciels** situés au dessus du système d'exploitation, comme les interpréteurs de commandes (shell), les compilateurs, les éditeurs, etc ...
     `Attention!` ces programmes ne font pas partie du système d'exploitation. En effet, un utilisateur peut réécrire un compilateur par contre il ne peut pas réécrire son propre gestionnaire des interruptions du disque car celui ci fait partie du système d'exploitation et est protégé.
3. Les programmes d'application sont situés au dessus du noyau. Ils sont écrits par les utilisateurs ou éditeurs de logiciels pour résoudre des problèmes spécifiques.

2.1.2 Les tâches du système d'exploitation

Le système d'exploitation effectue fondamentalement deux tâches sans rapport l'une avec l'autre.

Machine virtuelle L'architecture des ordinateurs au niveau du langage machine est primaire et lourde à programmer, en particulier en ce qui concerne les entrées/sorties.

Le système d'exploitation est le programme qui cache le matériel aux utilisateurs. Par exemple, il masque les éléments fastidieux liés aux interruptions, aux horloges, à la gestion mémoire, à la gestion des périphériques.

De plus, le système d'exploitation offre une même interface quelque soit le matériel. Par exemple, l'utilisateur accède de la même façon à des imprimantes de types différents.

Le système d'exploitation offre donc aux utilisateurs une machine étendue ou machine virtuelle plus facile à programmer que le matériel.

Gestionnaire de ressources Nous venons de présenter le système d'exploitation en tant qu'interface de programmation commode. Mais on peut aussi le considérer en tant que `gestionnaire de fonctions complexes`. Dans cette optique, le travail du système d'exploitation consiste à ordonner et à controler l'allocation : des processeurs, des mémoires et des périphériques entre les différents programmes qui y font appel.

Par exemple, si plusieurs programmes s'exécutant sur un même ordinateur essaient d'imprimer simultanément leurs résultats sur une même imprimante : il pourrait y avoir un mélange des résultats lors de l'impression. Pour éviter cela, le système d'exploitation a recours à une file d'attente et à un fichier tampon sur disque. De cette façon les résultats sont imprimés les uns après les autres, sans que les programmes s'en aperçoivent.

Lorsqu'un ordinateur est partagé entre plusieurs utilisateurs, la nécessité de gérer et de protéger la mémoire, les périphériques est encore plus évidente.

Dans ce contexte, le **rôle** du système d'exploitation est :
- de connaître à tout moment l'utilisateur d'une ressource,
- de gérer les accès à cette ressource,
- d'en accorder l'usage
- et d'éviter les conflits d'accès entre différents programmes ou entre utilisateurs.

2.1.3 Principes des systèmes d'exploitation

L'interface entre le système d'exploitation et les programmes de l'utilisateur est constituée d'un ensemble d'`appels systèmes`. Ces appels systèmes permettent de créer, détruire ou utiliser les objets logiciels gérés par le système d'exploitation, en particulier les processus et les fichiers.

Les processus

C'est le concept clé des systèmes d'exploitation. Un processus est un programme qui s'exécute. Il regroupe toutes les inforamations nécessaires à l'exécution d'un pro-

gramme. Citons quelques appels systèmes concernant les processus : création, fin de processus, demande de mémoire, etc.

Les fichiers

Un fichier contient des données, suite contigüe et non structurée d'octets. La gestion des fichiers par le système d'exploitation permet entre autre de masquer les spécificités des disques et autres périfériques d'entrée/sortie. En effet, on va pouvoir utiliser les périphériques d'entrée/sortie comme des fichiers spéciaux, c'est à dire lire et écrire dessus.

Les appels systèmes

Les appels systèmes étant le plus souvent écrits en assembleur, on associe à chaque appel système une fonction de bibliothèque (portant le même nom) que l'utilisateur peut appeler directement depuis son programme. Cette fonction de bibliothèque place les paramètres de l'appel système dans les registres du processeur puis exécute une instruction `TRAP` pour activer le système d'exploitation. La fonction de bibliothèque a donc pour buts de masquer les détails de l'instruction `TRAP` et de faire apparaître les appels systèmes comme des appels de procédures ordinaires.

Une instruction `TRAP` commute le processeur du mode utilisateur au mode noyau. La plupart des processeurs ont deux modes de fonctionnement pour des raisons de sécurité :

- le mode noyau ou super-utilisateur : dans ce cas, le processeur peut exécuter toutes les instructions et utiliser toutes les caractéristiques du matériel. Le système d'exploitation s'exécute dans ce mode.
- le mode utilisateur : où certaines instructions sont interdites (par exemple les instructions de gestion mémoire, des entrées/sorties). Les programmes utilisateurs s'exécutent dans ce mode.

L'instruction `TRAP` est une interruption. Rappelons que les interruptions conduisent à exécuter du code système et qu'elles sont de trois types :

- les exceptions : ce sont les erreurs d'exécutions (page fault, segmentation fault, etc).
- les interruptions matérielles (erreur bus, touches clavier, disques, périphériques, réseau, timmer, etc).
- les TRAPS, déroutement ou interruptions logicielles.

Pour gérer ces interruptions, on utilise une table des interruptions qui permet au système d'accéder au code à exécuter lorsqu'une interruption est levée. Dans le cas des TRAPs, le code correspondant à l'appel système invoqué est exécuté.

2.2 Le système UNIX

2.2.1 Historique

Unix est un système d'exploitation (Operating System) créé aux Etats Unis, au début des années 1970, chez Bells Laboratories de AT&T.

A l'origine écrit en assembleur, il a été réécrit en C (langage défini par les mêmes équipes et créé à la même époque), ce qui lui a permis d'être facilement portable.

Deux branches de développement UNIX virent le jour dans les années 1980 :
– AT&T : UNIX System V
– Université de Berkeley (Californie) : UNIX BSD
Actuellement, c'est le system V release 4 qui sert de référence à la plupart des Unix commercialisés.

Les grandes dates d'UNIX jusqu'au System 5 R4 :

	Berkeley University	**Bell Labs**		**AT&T**	**Autres**
1969/70	Phi 5 *(Thomson et Richie)*				
1973		V5	*multi tâches et utilisateurs noyau en C*		
1975	BSD 1.0	V6	*commercialisation*		IDRIS
1978	BSD 2.0	V7	*File system définitif Portabilité du langage*		ONIX,MIMOS,ZEUS
1981	BSD (3.0, 4.0) $\rightarrow$ 4.1		*pipes nommés*	system 3	XENIX, VENIX
1983	BSD 4.2		***IPC***	system V	SUN,QNIX,ULTRIX
				system V R1	SPIX
		V8	*caractères 8 bits*	system V R2	ULTRIX32
1986	BSD 4.3		*sysadm, posix, internationalisation*	system V R3	SUNOS 4.0
1990			*système unifié (System V + BSD+XENIX) , C-Ansi (ISO) -Interface graphique*	system V R4	AIX, SOLARIS

2.2.2 Caractéristiques principales

UNIX est un système multi-utilisateurs : plusieurs utilisateurs peuvent ouvrir des sessions en même temps sur une machine.

C'est aussi un système multi-tâches : plusieurs programmes peuvent s'exécuter en même temps sur une machine en temps partagé.

UNIX offre la possibilté de travailler en réseau : transfert de données, connections distantes.

2.2.3 Les différents UNIX ou "unices"

Les UNIX commercialisés et/ou utilisés dans les entreprises sont principalement :
- Solaris société SUN
- AIX : société IBM
- HP-UX : Hewlett-Packard
- citons aussi OSF/1, ULTRIX, Spix, BOS/X ..

2.2.4 De quoi est constitué un UNIX ?

Unix est constitué de :
- un **noyau** qui est le système d'exploitation proprement dit, le programme de base encore appelé en anglais "kernel" : c'est ce noyau qui gère la mémoire, les entrés/sorties , l'ordonnancement des processus, l'accès aux périphériques. Il est lancé lorsque l'ordinateur est mis sous tension.
- un **interpréteur de commandes**, le "shell" : c'est l'interface noyau/utilisateur.
- des **utilitaires** : des outils pour vous permettre de travailler : compilateurs pour de nombreux langages, traitements de texte, messagerie électronique, ...

2.2.5 Linux

En 1984, est née l'idée d'un système d'exploitation libre. En 1985, la Free Software Fondation (FSF) est crée pour fournir un cadre juridique à ce projet. Les bases de l'environnement ont été définies, puis des outils ont été développés.

Linux, système Unix libre sur plate forme PC était au départ un projet de loisir d'un étudiant finlandais : Linus Torvalds. Linux fut inspiré de Minix, un petit système Unix développé par A.Tannebaum.

Fin 1992, la première version officielle de Linux voit le jour. Puis Linux évolue grâce à son concepteur mais aussi à tous ses utilisateurs qui peuvent participer à son développement et aussi au développement d'outils. Linus Torvalds a accepté que le code de Linux soit sous licence GPL (General Public Licence).

Linux est une libre implémentation des spécification POSIX avec des extensions System V et Berckeley.

Linux est le plus souvent diffusé sous forme de distributions : un ensemble de programmes formant après installation un système complet.

Il existe plusieurs versions de Linux : Debian, Mandriva Linux (ex Mandrake, société Mandriva), RedHat/Fedora (distribution communautaire sponsorisée par Red Hat), SlackWare, SuSE (société Novell), Ubuntu.

2.3 Premiers pas avec le système UNIX

2.3.1 Se connecter

Pour entrer dans un système multi utilisateurs, il faut se faire connaître par un identifiant et un mot de passe pré définis dans le système. Sous Unix, la distinction est faite entre les minuscules et les majuscules, que ce soit pour le login ou pour les noms de fichiers ou pour les commandes. **Bonjour** ne sera pas équivalent à **BONJOUR** ni à **bonjour**.

Login :

A cette invite, vous tapez le nom de login qui vous a été attribué

Password :

Par sécurité, l'ouverture d'une session est protégée par un mot de passe, à renseigner en veillant à bien respecter les majuscules/minuscules.

Identification incorrecte

Le système me renvoie le message suivant :

```
login incorrect
login:
```

Identification correcte

Le système me renvoie un message de type

```
Last login : We Sept 15th 12:30:15
```

A partir de ce moment, vous êtes connecté. Vous êtes en cours d'exécution de votre premier programme , le shell. C'est ce programme qui vous permet d'entrer des commandes en ligne. La première ligne indique la date et l'heure de votre précédent login. Cette ligne est utile afin de s'assurer que personne d'autre n'a utilisé votre login. Le "@" est le prompt (ou invite) d'accueil. Nous verrons qu'il peut être paramétré, le votre pourra donc être différent. Dans la suite du cours le prompt pourra être initialisé à $. Vous pouvez rencontrer avant le prompt une phrase qui est maintenue dans un fichier `motd` (message of the day). Cette phrase peut servir par exemple à informer les utilisateurs du système d'une intervention sur la machine ... ou tout autre information !

Terminer sa session utilisateur

La déconnexion se fait en entrant la commande `exit`.
Le système renvoie un message de type :

Listing 2.1– Terminaison de session

```
@ exit
Last login : We Sept 15th 12:30:15
@
```

Changer son mot de passe

Lors de la première connexion, le système vous demande de changer votre mot de passe. A tout moment, il est possible de modifier son mot de passe en entrant la commande :

Listing 2.2– Changement de mot de passe

```
@ passwd
changing password for user
new password:
reenter new password:
@
```

Arrêter le système

Pour arrêter le système, l'administrateur lance l'une des commandes suivantes :

Listing 2.3– arrêt immédiat

```
\shell{halt} (= \shell{shutdown -h now})
```

Listing 2.4– arrêt différé

```
\shell{shutdown -h nn}
```

il s'écoulera `nn` *minutes entre l'avertissement et l'arrêt*

Listing 2.5– pour un reboot

```
\shell{shutdown -r now} ou \shell{reboot} ou \shell{ctrl-alt-del}
```

C'est à l'invite du message "The system is halted" que l'on peut éteindre électriquement sa machine.

2.3.2 La ligne de commande

La commande

A partir du prompt, vous allez entrer des commandes. Pour que la commande soit prise en compte par le shell, elle doit être suivie de la touche ENTER.
La ligne de commande Unix est structurée de la façon suivante :
@ commande [-options] [argument1] .... [argument n]

où :
– les séparateurs utilisés sont des espaces (un ou plusieurs).
– les options de la quasi totalité des commandes sont préfixées par un tiret (-).
– les crochets [ ] entourent les termes optionnels

Listing 2.6– Exemple de lignes de commande

```
@ wc /etc/passwd
15 32 989 /etc/passwd
```

compte les lignes, mots et caractères dans le fichier "/etc/passwd" : ce fichier comporte 15 lignes, 32 mots et 989 caractères

Interruption d'affichage de commande

L'affichage d'une quelconque commande peut être interrompu de la manière suivante :

```
@ commande | pg
```

ou

```
@ commande | more
```

Le résultat est affiché page par page. L'appui sur la barre d'espace avance d'une ligne, l'appui sur la touche <enter> avance d'une page (= le nombre de lignes d'un écran).

⇒La commande "commande" envoie ses informations à la commande "pg" (page) ou "more" (plus) par l'intermédiaire d'un pipe, ou tube de communication que nous verrons dans le chapitre 3 "Les filtres" .

L'arrêt du défilement peut aussi être activé pendant l'exécution d'une commande par CTRL-S et le défilement repris par CTRL-Q.

Les messages d'erreurs

Le message peut sensiblement différer d'un Unix à l'autre, ce qui se comprend à la lecture de l'historique d'Unix. Néanmoins, un anglais rudimentaire permet de comprendre rapidement le message reçu.

Les messages donnés en illustration sont ceux des Unix "Linux Red Hat" (L) , "Sun OS" (S) et/ou "AIX" (A)

1. La commande n'existe pas ou n'est pas trouvée :

Listing 2.7– Messages d'erreur : commande non trouvée

```
@ toto
bash:toto:not found (L)
```

2. La commande est erronée : chaque commande vérifie ses arguments et affiche son message d'erreur. Dans la plupart des cas, le message d'erreur affiche la syntaxe correcte de la commande.

Listing 2.8– Messages d'erreur : commande erronée

```
@ wc -y /etc/group
usage:wc [-c|w] [name ...]
```

ou

Listing 2.9– Messages d'erreur : commande erronée

```
@ wc -y /etc/group
wc:invalid option --y
Pour en savoir davantage, faites : 'wc --help'
```

ou

Listing 2.10– Messages d'erreur : commande erronée

```
@ wc -y /etc/group
wc:illegal option --g}

 usage: wc[-c|-m|-C] [-lw] [file ....]
```

2.4 La documentation en ligne

La documentation sur toutes les commandes et leur syntaxe est accessible par la commande :

```
@ man [numéro de chapitre] commande
```

`Man` est disponible sur tous les Unix, et organisé de la même façon.

La documentation est organisée en 3 chapitres :
– chapitre 1 : la plupart des commandes
– chapitre 2 : les appels systèmes
– chapitre 3 : les bibliothèques de langages
Le numéro de chapitre n'est pas obligatoire, il permet simplement d'éviter la recherche dans tous les chapitres.

La documentation est présentée pour chaque commande de la façon suivante :

1. un texte, en anglais ou parfois en français (sous Linux notamment), expliquant le but de la commande.
2. la liste des paramètres de la commande avec pour chacun le détail de ce qu'il permet.
3. des exemples avec les divers paramètres dont vous pouvez vous inspirez !
4. la rubrique SEE ALSO donne la liste des commandes ayant un rapport avec la commande recherchée, ce qui permet éventuellement de trouver une commande plus appropriée à son besoin.

Déplacement

Le déplacement page avant , recherche de caractères etc... dans le `man` se fait à l'aide des commandes "vi" :

- pour avancer d'une page à l'écran appuyer sur la barre d'espace,
- pour avancer d'une ligne appuyer sur $< enter >$,
- pour sortir tapez `q` (comme quit).

⇒ **Ne pas hésiter à user du "man" ! ! !**

2.5 L'éditeur de texte Vi

Vi (prononcer "vi aïe") est un éditeur de texte qui permet de travailler en mode écran. Cet éditeur de texte est disponible sur tous les systèmes Unix.

Les différents modes

Vi travaille en deux modes :
– le mode commande
– le mode insertion

Toutes les commandes se passent à l'aide du clavier, ce qui signifie que "appuyer sur la touche `a`" n'a pas le même effet selon que l'on se trouve en mode commande ou en mode insertion.

On passe du mode commande au mode insertion à l'aide de la touche ESC (échappement).

Dans le mode commande, les commandes qui commencent par **:** doivent être validées par $< enter >$, les autres sont excécutées immédiatement.

Lancement de vi :

Pour lancer l'éditeur , entrer la commande :

```
@ vi [+commande] [fichier]
```

Au lancement, vi est en mode commande.

L'option `+commande` permet à l'ouverture d'excécuter une commande ligne quelconque. La plus courante utilisation est de se positionner directement sur une ligne du fichier.

Listing 2.11– Commande vi

```
@ vi +15 monfichier
```

permet d'ouvrir le fichier "monfichier" en se positionnant directement à la ligne 15.

Commandes de sortie :

Pour quitter vi, différentes possibilités sont offertes.

Commande	Action
:w [mon_fichier]	vide le buffer dans le fichier en cours si aucun nom n'est indiqué, dans mon_fichier sinon
:q	quitte sans rien écrire
:q !	Le ! permet de quitter si le buffer a été modifié mais pas sauvegardé
:wq	vide le buffer dans le fichier en cours puis quitte
:x	équivalent à `:wq`
ZZ	équivalent à `:wq` mais sans avoir à faire $< enter >$

Commandes d'annulation :

Elles sont au nombre de deux :

Commande	Action
`u`	Permet d'annuler le dernier changement
`:e!`	Annule les modifications et reprend la dernière version enregistrée.

Commandes de déplacement par mot et par écran

Les commandes de déplacement du curseur par mot et par écran : le [n] qui précède une touche indique la répétition de cette touche.

Touche	Quelle action ?
[n]w	curseur sur le mot suivant
[n]b	curseur sur le mot précédent
[n]e	curseur à la fin du mot courant
[n]H	curseur sur la 1ère ligne de l'écran n (Hight)
[n]L	curseur sur la dernière ligne de l'écran n (Low)
M	curseur au milieu de l'écran (Middle)
Ctrl-D	déplacement d'un demi écran vers l'avant
Ctrl-U	déplacement d'un demi écran vers l'arrière
Ctrl-F	déplacement d'un écran vers l'avant (Forward)
Ctrl-B	déplacement d'un écran vers l'arrière (Back)
z	met la ligne courante au centre de l'écran

Commandes de déplacement par caractère et par ligne :

Le `[n]` qui précède une touche indique la répétition de la touche (ne se trouvent ici que les commandes les plus usitées).

Touche	Quelle action ?
[n]h	déplacement vers la gauche
[n]l	déplacement vers la droite
[n]k	déplacement vers le haut dans la même colonne
[n]j	déplacement vers le bas dans la même colonne
[n]+	curseur sur le nième premier caractère de la ligne suivante
[n]	curseur sur le nième premier caractère de la ligne précédente
0	curseur en début de ligne
∧	curseur sur le premier caractère non blanc de la ligne
$	curseur en fin de ligne
[n]G	aller à la ligne n (G = aller à la dernière ligne)
[n]I	curseur en colonne n
fy	curseur sur le prochain caractère y
Fy	curseur sur le précédent caractère y
tx	curseur devant le prochain caractère x
Tx	curseur devant le précédent caractère x
mz	marque la position courante avec la lettre "z" (lettres de a à z)

Mode insertion :

La sortie du mode insertion se fait en appuyant sur la touche ESC (ne se trouvent ici que les commandes les plus usitées).

Touche	Quelle action ?
a	insertion de texte après le curseur "append"
A	insertion de texte à la fin de la ligne courante
i	insertion de texte avant le curseur "insert"
I	insertion de texte au début de la ligne courante
o	insertion de lignes en dessous de la ligne courante
O	insertion de lignes au dessus de la ligne courante
Ctrl-H	efface le caractère précédent le curseur
Ctrl-W	efface le mot précédent le curseur
Ctrl-U	efface tous les caractères de l'insertion en cours

Modification sans changement de mode

(Ne se trouvent ici que les commandes les plus usitées)

Touche	Quelle action ?
rz	remplace le caractère sous le curseur par le caractère "z"
/motif	recherche motif vers l'avant
n	répète la dernière commande de recherche
~	change le caractère sous le curseur de majuscule en minuscule
[n]x	supprime n caractères à partir du curseur (1 par défaut)
[n]X	supprime n caractères avant le curseur (1 par défaut)
[n]dd	supprime n lignes à partir du curseur
[n]dw	supprime n mots à partir du curseur
dfz	supprime de la position du curseur jusqu'au 1er caractère "z" rencontré
D	supprime de la position du curseur jusquà la fin de la ligne
[n]yy	copie n lignes à partir du curseur
[n]yw	copie n mots à partir du curseur
p	colle les lignes ou mots stockés par "y" après la ligne ou le curseur
P	colle les lignes ou mots stockés par "y" avant la ligne ou le curseur

Modification avec changement de mode

Nous rappelons que la sortie du mode insertion se fait en appuyant sur la touche ESC.

Touche	Quelle action ?
R	change les caractères à partir du curseur
[n]s	substitue les n caractères par un nombre quelconque de caractères
[n]S	substitue toute la ligne
[n]cc	change les n lignes à partir du curseur
[n]cw	change les n mots à partir du curseur
[n]C	change du curseur à la fin de la ligne

2.6 Exercices

Le premier chapitre étant un chapitre de présentation, les exercices sont plus une exploration et une découverte à faire par vous même que de réelles questions / réponses.

1. Créer à l'aide de l'éditeur Vi un fichier nommé `monPremierFichier`.

   Le but de cet exercice est de vous familiariser avec les commandes vi : entrez un texte libre et modifiez le par ajout, suppression, modification de lignes, de caractères etc .... la seule limite étant votre imagination.

   Remarque :
   Si vous bootez sur le CD ROM, munissez vous d'une disquette formatée DOS que vous mettrez dans votre lecteur. Pour enregistrer puis re-travailler sur le fichier , vous le référencerez dans toutes les commandes comme s'appelant `/mnt/floppy/monPremie`

2. Appliquez au fichier `monPremierFichier` la commande `wc` après chaque modification/enregistrement de votre fichier et vérifiez le résultat.

3. Consultez la documentation sur vi, wc, passwd.

Chapitre 3

Gestion des fichiers -GL-

Dans ce deuxième chapitre, vous vous familiariserez avec la gestion des fichiers sous UNIX

3.1 Unix et le stockage des données

3.1.1 Fonctionnement d'un disque dur

En préalable, revenons sur quelques notions physiques sur les disques, ceci afin de mieux comprendre les principes de gestion des fichiers.

Un disque est un ensemble de plateaux magnétiques entre lesquels se déplacent des têtes de lecture. Les disques tournent autour de leur axe, les têtes se déplacent latéralement, de chaque côté des plateaux, et permettent de lire ou d'écrire l'information.

Chaque disque comporte n *pistes* , une piste étant la partie lue par une tête de lecture immobile lorsqu'un qu'un disque effectue une révolution complète.

Le *cylindre* est l'ensemble des pistes de même diamètre sur les disques qui sera lu à un instant t par l'ensemble des têtes de lecture.

Le *secteur* est une subdivision régulière des pistes. C'est la plus petite partie qui sera lisible. Comme tous les disques n'ont pas la même taille de secteur, le système utilise la notion de *blocs* pour s'affranchir des contraintes physiques liées à chaque constructeur. Un bloc est un multiple entier de secteurs.

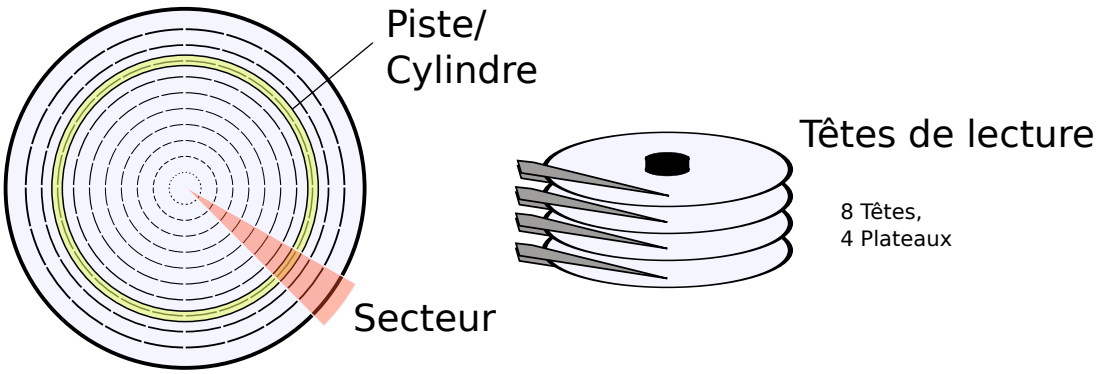

FIGURE 3.1 – Structure d'un disque dur

3.1.2 Formatage d'un disque dur

Le contrôleur de disques durs gère les disques et assure la liaison entre le processeur et le disque dur. Pour être utilisable par un système d'exploitation, le disque doit être préparé : c'est le formatage. Deux formatages :

- Le formatage "bas niveau" (formatage physique) consiste à diviser le disque en pistes, secteurs et cylindres. Les pistes sont numérotées en commençant par 0 ; chaque piste est divisée en secteurs numérotés en commençant par 1. A cette étape, le système d'exploitation n'a pas d'influence, le but est de préparer physiquement le disque et de détecter les éventuels secteurs défectueux. Ce formatage est réalisé sur les disques durs du commerce lors de leur fabrication en usine.
- Le formatage "haut niveau" consiste à créer les structures de gestion des fichiers (formatage logique) C'est lors de ce formatage que l'on précise le nom du *système de fichiers* qui sera utilisé pour cette partie du disque. Le type de système de fichiers dépend du système d'exploitation qui devra l'exploiter. Chaque système d'exploitation demandant généralement une partition dédiée, il est possible de partager le stockage total du disque dur en partitions, qui pourront utiliser des systèmes de fichiers différents pour chaque système d'exploitation installé, comme nous le verrons dans la section suivante.
  Le tout premier secteur d'un disque dur contient les informations sur le disque dur. S'il contient le code qui servira à amorcer le système, il prend le nom de "Master Boot Record". Ce secteur est sans doute le plus important puisqu'une fois détruit, le système ne pourra plus s'amorcer à partir de votre disque, et la structures des partitions présentes sur le disque dur sera perdue.

3.1.3 Partitionnement d'un disque dur

Lors de son achat, un disque dur est livré avec une structure totalement vierge : Avant de pouvoir le formater logiquement avec un système de fichier, il est nécessaire

de le partitionner, c'est à dire de définir la manière dont sera réparti l'espace total de stockage du disque en différentes zones pouvant accueillir des données, ou *partitions*.

Ce partitionnement est réalisé à l'aide d'outils dédiés, disponibles sous la plupart des systèmes d'exploitation, tels que `fdisk`, `cfdisk` ou `parted` sur un système d'exploitation UNIX. Il est destructif : toute ancienne partition, et les données qu'elle contenait, ne pourra plus être récuperée après la partition, sans utilisation de méthodes complexes et d'équipements coûteux.

La plupart des disques durs modernes utilisent un partitionnement de type PC, permettant la création d'au maximum quatre partitions dites primaires, c'est à dire directement enregistrées dans la table des partitions stockée sur le premier secteur du disque dur.

Cependant, pour contourner cette limitation en terme de nombre de partitions, il est possible de définir la quatrième partition primaire comme étant une partition de type étendue : Cette dernière partition comprendra alors sa propre table de partition, permettant la création d'une ou de plusieurs partitions secondaires dans la zone du disque dur correspondant.

Certains systèmes, comme Microsoft Windows, préfèrent démarrer sur la première partition primaire du disque dur. D'autres, comme Linux, n'imposent pas ces contraintes : en règle générale, chaque système d'exploitation aura ou non des exigences à ce niveau.

3.1.4 Système de gestion de fichiers UNIX

Le système de gestion de fichiers consiste en l'ensemble des primitives qui seront utilisées pour localiser, accéder, gérer les données sur les supports. Nous allons voir que sous UNIX, le fichier n'est pas seulement un fichier disque. C'est un objet dont le type détermine les actions qui pourront lui être appliquées. Le système de gestion des fichiers est indépendant de la structure physique des supports. Il regroupe les primitives systèmes qui permettent l'accès à ces différents types de fichiers.

Un fichier peut être associé à :
- un disque physique
- un terminal
- une ressource physique du système (mémoire)
- une ressource logique du système (partition logique)
- ...

Ses caractéristiques principales sont :
- Structure arborescente (la racine est représentée par le symbole /)
- Possibilité d'avoir plusieurs systèmes de gestion de fichiers différents
- Protection des fichiers simple et applicable à tous les systèmes de fichiers
- Possibilité de désigner un même fichier physique par plusieurs noms dans des systèmes de fichiers différents

- Extension de l'arborescence dynamique par mécanisme de montage
- Les systèmes de fichiers doivent se "monter", ce qui correspond à assigner un répertoire à une structure physique
- Un système de fichiers peut occuper :
  1. un volume physique
  2. une partition d'un volume physique
  3. une partition d'un groupe de volumes physiques

L'accès aux support se fait par blocs, unité de manipulation du système de gestion de fichiers : plus le bloc sera grand, plus la place perdue sera importante : si la taille du bloc est par exemple de 4 Ko (taille pour NTFS), un fichier même vide occupera 4 Ko puisque c'est la plus petite unité d'allocation du système de fichiers ; si la taille est de 512 o , on aura une meilleure utilisation de la surface, mais une plus grande proportion de fichiers dispersés sur plusieurs blocs (fragmentation).

Nous allons développer toutes ces notions dans la suite de ce chapitre.

3.1.5 Systèmes de fichiers

Comme nous l'avons vu précedemment, une partition de disque dur ne représente qu'un ensemble de secteurs magnétiques non structuré. Pour pouvoir stocker le contenu des fichiers et les informations qui leurs sont associées (taille, propriétaire, permissions, etc), il est nécessaire de regrouper ces secteurs dans une structure logique, le système de fichier, dont le type définit l'utilisation dévolue à chaque emplacement.

Il existe de nombreux systèmes de fichiers, dédiés à des utilisations spécifiques (base de données, par exemple), à certains types de supports, ou à un système d'exploitation et ses fonctionnalités : chaque partition ne peut contenir qu'un seul système de fichier.

Voici une liste de quelques systèmes de fichiers disponibles sur les Unix modernes :

nfs network file system

jfs journalized file system

hfs+ système de fichiers Mac OS X

ext2 système de fichiers de linux

ext3 amélioration du format ext2

vfat système de fichier FAT 32

proc visualisation et manipulation des processus

fd descripteurs de fichiers (/dev/fd)

La liste des systèmes de fichiers supportés par le système de travail se trouve dans /proc/filesystems, et la commande `fsck` permet de vérifier et de réparer les systèmes de fichiers endommagés (cohérence entre les fichiers présents et la table des inodes, références vers les blocs de données...)

Sous UNIX, tout est fichier : certains systèmes de fichiers sont ainsi purement virtuels, et ne correspondent à aucune partition. Ils servent de représentation logique à un aspect donné du système, tel que les processus en cours d'exécution, ou les périphériques disponibles.

Deux exemples de tels systèmes de fichiers sont `fd` et de `proc` :

- `proc` représente ainsi les processus actifs et leurs différentes propriétés (arguments, répertoire de travail...) sous formes de fichiers.
- `fd` permet d'accèder aux descripteurs de fichiers[1]. Ceci permet de simplifier la programmation car à l'intérieur des programmes ces descripteurs sont vus comme des fichiers

Ces types de systèmes de fichiers n'occupent aucune place sur le disque dur, et sont émulés dynamiquement à chaque accès par le noyau du système d'exploitation.

Les inodes

Sur un système UNIX, tout fichier est avant tout stocké sous la forme d'un noeud d'index, ou *inode*, contenant l'ensemble des métainformations associées aux fichiers.

Ce noeud d'index comprend également plusieurs pointeurs vers les zones du disque dur contenant le contenu du fichier à proprement parler, sous forme de séquences de secteur linéaires : Si ces zones ne sont pas contigües, on parlera de fragmentation.

Le noeud d'index ne contient pas le nom du fichier, mais un entier permettant de l'identifier de manière unique au sein du reste du système de fichier, appellé numéro d'inode : plusieurs noms peuvent représenter des liens vers un même fichier, comme nous le verrons plus loin.

Structure d'un inode en Unix système V

Un noeud d'index UNIX comprend, quelle que soit le type de système de fichiers, les informations suivantes :

- type du fichier et permissions
- propriétaire
- groupe
- taille
- nombre de liens
- date de dernier accès

1. Nous reparlerons des descripteurs au chapitre 4 : redirection des entrées-sorties

– date de dernière modification
– date de dernière modification de l'inode

Les dates sont par convention stockées sont la forme du nombre de secondes écoulées depuis le 1er janvier 1970 (année "officielle" de la naissance d'Unix).

Remarque : Sur la plupart des systèmes d'exploitation Unix 32 bits traditionnels, cette date est stockée sous forme d'entier, ce qui ne permet pas de gérer une date au delà du 19 janvier 2038 : Cette limitation n'est plus valable sur la plupart des Unix récents ou les systèmes 64 bits.

Représentation physique d'un système de fichiers

Le système de fichiers est constitué :

– d'un super bloc dans lequel on trouve les caractéristiques du bloc de fichiers. Ce super-bloc est lu au montage du système de fichiers et mis à jour régulièrement sur le disque : il contient les informations du système de fichiers dont la I-List (inode Liste) correspondant à la liste des inodes, propre à un système de fichier donné. Ainsi, dans un système de fichiers donné, tout numéro d'inode est unique.
– les blocs de données proprement dits.

Dans la section suivante, nous donnons un exemple d'accès à un fichier à l'aide de la I-LIST.

3.1.6 Organisation des fichiers sous Unix

La majorité des ordinateurs contiennent plusieurs sources de données, telles qu'un lecteur CDROM, une ou plusieurs partitions, ou des partages réseaux, chacun de ces périphérique comprenant un ou plusieurs systèmes de fichiers.

Contrairement aux systèmes de type Windows, qui assignent une lettre à chacun de ces volumes de données, UNIX privilégie l'abstraction du stockage : l'ensemble des fichiers est organisé au sein d'une seule arborescence unifiée même si celle-ci regroupe le contenu de plusieurs périphériques physiques, et donc plusieurs systèmes de fichiers. Son répertoire de plus haut niveau est nommé répertoire *racine* (*root*) représenté par "/".

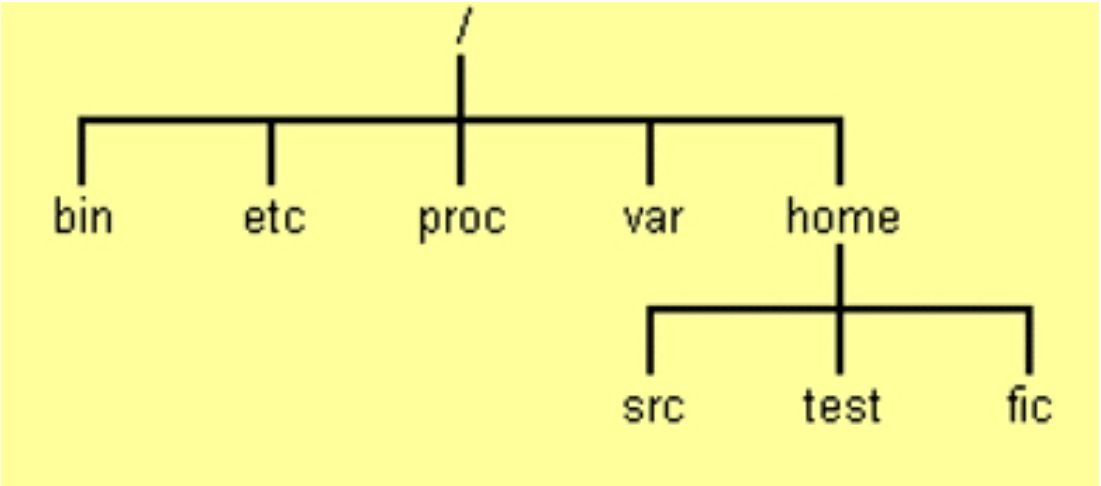

FIGURE 3.2 – Structure arborescente

Chemin d'accès à un fichier

Pour accèder à un fichier, on précise son chemin à l'intérieur de cette arborescence. Il existe deux sortes de chemins :

- Le chemin absolu, partant de la racine.
- Le chemin relatif. Tout processus Unix est associé à un répertoire de travail (working directory), ou répertoire courant, qui servira alors de référence pour l'interprétation de ce type de chemin, à la place de la racine.

Exemples : de chemin absolu et relatif équivalents, si on se trouve dans le répertoire `/home/mesfichiers`, ce dernier répertoire est appelé *répertoire courant* ou *répertoire de travail* :

- Chemin d'accès absolu : `/home/mesfichiers/travail/fichier1`
- Chemin d'accès relatif : `travail/fichier1`

Tout répertoire Unix comprend deux entrées spéciales, qui représentent des liens permettant de remonter dans le système de fichier :

- le répertoire . qui représente le répertoire lui même
- le répertoire .. qui représente son répertoire parent

Les chemin d'accès : `travail/toto` et `./travail/toto` sont équivalents. Lorsque nous traiterons les variables, nous verrons une utilisation de la seconde notation.

Exemple d'accès à un fichier par le système

Dans l'exemple ci dessous est donnée la reprsentation physique permettant l'accès au fichier : `/toto/titi/fichier`.

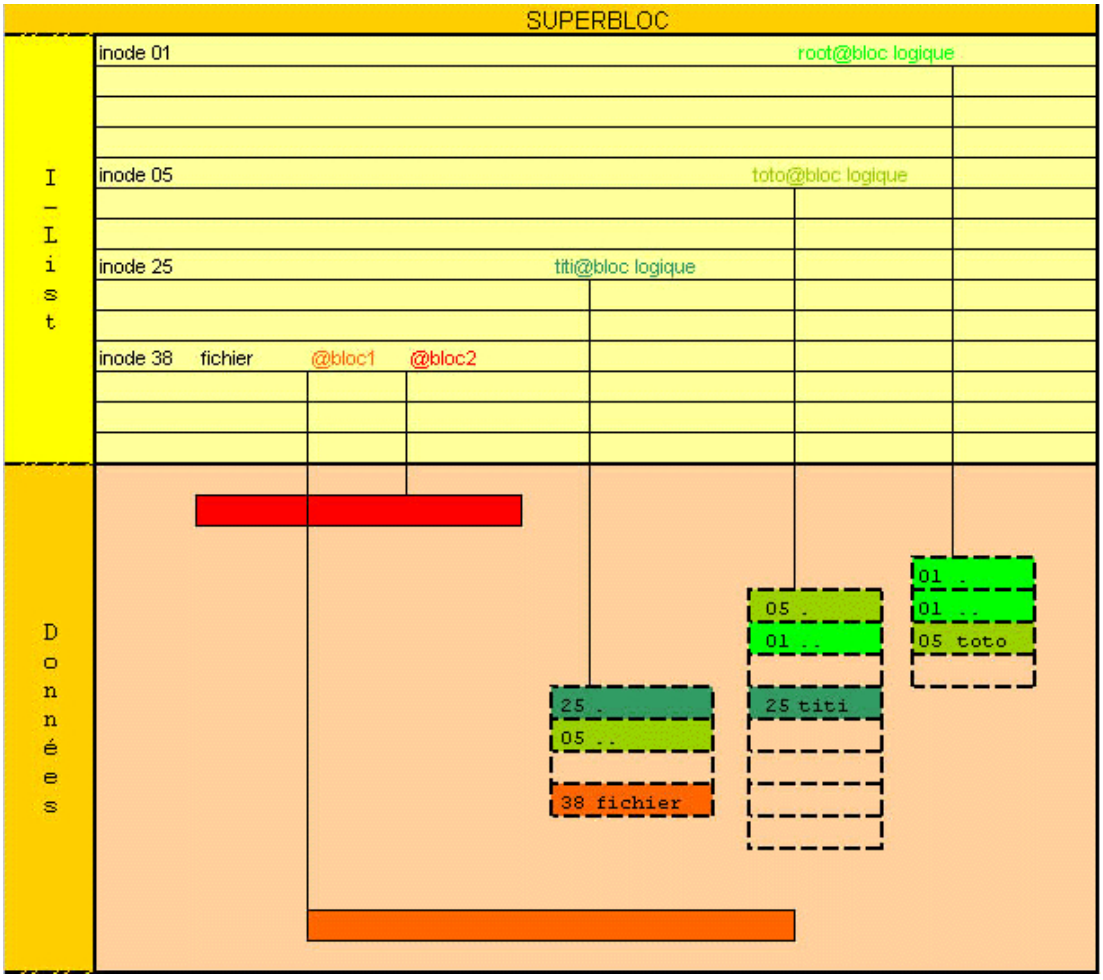

FIGURE 3.3 – Représentation physique du système de fichier illustrée par un exemple

Ce schéma illustre comment le système accède à un fichier à partir d'un chemin d'accès absolu.

1. Après la recherche de la racine inode 1, il trouve l'adresse à laquelle seront stockées les données contenues dans ce répertoire (adresse appelée root@bloclogique). Ce répertoire contient
   – le numéro d'inode de lui même (.) : 01
   – le numéro d'inode de son père (..) : 01 et
   – le numéro d'inode de toto : 05.
   L'inode 1 représentant la racine, son père est lui même .
2. Le système va alors au numéro d'inode 05 (toto), à l'adresse de ses données : la lecture des données indique qu'il s'agit à nouveau d'un répertoire car il contient des numéros d'inodes :
   – le numéro d'inode de lui même : 05,
   – le numéro d'inode de son père : 01 (la racine) et
   – le numéro d'inode de titi (25).

3. Même cheminement pour le numéro d'inode 25 qui contient entre autres le fichier de numéro d'inode 38.
4. On atteint le numéro d'inode 38 ; on peut accéder à ses données car dans l'inode on a la liste des blocs de données correspondants au fichier. Ici , deux blocs : @bloc1 et @bloc2. Le fichier est donc de taille comprise entre 1 et 2 blocs .

3.1.7 Structure arborescente standard UNIX

Sur un système Unix standard, on trouve les répertoires suivants :

/bin Commandes de base du système

/sbin Commandes de base de l'administrateur

/dev Pseudo-fichiers permettant de communiquer avec les périphériques

/lib Bibliothèques de base du systèmes et modules du noyau

/mnt Points de montages externes

/tmp Fichiers temporaires

/etc Fichiers de configuration

/usr Programmes utilisateurs

Les répertoires suivants sont optionnels :

/home Contient par convention les répertoires personnels utilisateurs

/usr/local Programmes compilés et installés en local

Le standard d'organisation FHS (Filesystem Hierarchy Standard) tend à améliorer et uniformiser ces répertoires et leur contenu entre la plupart des Unix modernes, à quelques exceptions et divergences près.

Remarque : Sur le systèmes de type KNOPPIX, "/" n'est en fait qu'une majorité de liens vers le répertoire /KNOPPIX localisé sur le CDROM.

3.1.8 Notion de montage

Comme il n'existe qu'une seule arborescence de fichier sous UNIX, il est nécessaire de combiner les différentes sources de données au sein de celle-ci, ce qui se traduit par l'association de la racine de chaque système de fichier à un répertoire donné de l'arborescence existante (par exemple `/home`) : Ce processus est nommé *montage*.

Pour disposer des répertoires nécessaires à ce montage, la racine "/" correspond à une partition principale spécifiée au démarrage du système, nommé partition racine, chargée de fournir les points de montage pour le reste des systèmes de fichiers et de contenir la partie de l'arborescence système non fournie par ceux-ci.

La commande mount

Par défaut, l'opération de montage ne peut être réalisée par un simple utilisateur, mais uniquement par l'administrateur, sauf s'il en accorde la permission à d'autres utilisateurs ou groupes d'utilisateurs, et s'effectue à l'aide de la commande suivante :

```
mount –t t_syst_fich /dev/nom_dev /mnt/rép_acc –o opt_montage
```

- `t_syst_fich` : type de système de fichiers utilisé
- `/dev/nom_device` : nom du device à monter
- `/mnt/rép_accueil` : répertoire dans lequel monter le device
- `options_montage` : options de montage (décrites par la suite)

Fichier /etc/fstab

Le fichier `fstab` sert à mémoriser les informations des principaux montages utilisés par le système. Si le point de montage est enregistré dans le fichier `/etc/fstab`, la commande de montage devient donc plus simple :

```
mount chemin_du_device_ou_point_de_montage
```

Un extrait de fichier `fstab` est donné ci-dessous :

Listing 3.1– Extrait de fichier `/etc/fstab`

```
# 1: device   # 2: mountpoint   # 3: type   # 4: options    # 5,6: flags
/dev/sda1     /                 ext3        relatime         0       1
/dev/sda8     /home             ext3        relatime         0       2
/dev/sda9     /opt              ext3        relatime         0       2
```

Les informations y sont organisées de la manière suivante, séparées par un ou plusieurs espace ou caractères de tabulation :

1. nom du périphérique à monter
2. nom du point de montage
3. type de système de fichiers
4. options de montage : permissions par défaut, encodage des caractères, montage automatique ou non au démarrage, gestion des quotas...
5. indicateur de prise en compte par les outils de sauvegarde "dump" (0 : Point de montage ignoré, 1 sinon)
6. ordre de vérification des systèmes de fichiers (0 : Pas de vérification, 1 : Prioritaire, 2 : Normal)

le fichier `/etc/mtab` respecte le même format mais correspond aux systèmes de fichiers actuellement montés.

3.1.9 Les noms de fichiers

Les noms de fichiers peuvent comprendre de 1 à 255 caractères, accentués ou non, à l'exception de "/" et NULL, marquants respectivement un niveau d'arborescence et la fin de nom de fichier. Il ne s'agit là que des limitations imposées au niveau du système, et le système de fichier utilisé pour le stockage ajoute parfois des contraintes supplémentaires au nommage.

Quelques noms de fichier corrects :

- 5
- fichier
- fichier_1
- fichier.backup
- .ficher_caché

Le préfixe "." donné à un nom de fichier fait de celui-ci un fichier caché, c'est à dire non visualisable par la commande standard de listage d'un répertoire. Dans le listing précédent, `.fichier_caché` est ainsi un fichier caché. Il s'agit d'une convention des outils Unix : Il est possible de demander l'affichage de ces fichiers à la commande `ls` ou au navigateur de fichiers et il s'agit de fichiers tout à fait normaux par ailleurs.

Attention, il est possible d'utiliser des caractères tels que , & dans les noms de fichiers, qui sont par défaut interprétés par le shell : Dans ce cas, il faudra protéger le nom de ces fichiers en l'entourant de guillemets (") ou de quotes (').

Les caractères spéciaux : Certaines commandes UNIX s'appliquent à plusieurs fichiers. Dans ce cas on utilise un nom de fichier générique construit à partir de caractères spéciaux ou métacaractères remplaçant un ou plusieurs caractères dans le nom de fichiers.

- * remplace n'importe quelle chaîne de caractères,
- ? n'importe quel caractère,
- `[ ]` un caractère parmis ceux qui se trouvent entre les crochets,
- `[ - ]`un caractère dans un intervalle de caractères avec les crochets,
- ! en première position dans les crochets signifie la négation des caractères présents.

Exemples :
- *.* : *tous les noms de fichiers dont le nom comprend un point (pas en 1ère position)*
- * : *tous les fichiers en parcourant l'arborescence*
- f* : *tous les fichiers dont le nom commence par f*
- f ? : *tous les noms de fichiers composés de f et n'importe quel caractère*

- f[12xy] : *un (seulement) nom de fichier de la forme : f1 ou f2 ou fx ou fy*
- f[a-z] : *fa ou fb ou ... fz*
- *.c : *tous les noms de fichiers se terminant par .c*
- ?.c : *tous les noms de fichiers composés de un caractère et .c*
- ? ? : *noms de fichiers de 2 caractères*
- *.[a-zA-Z] : *tous les noms de fichiers se terminant par . et un caractère*
- *.[ch0-9] : *tous les noms de fichiers se terminant par . et c ou h ou un chiffre*
- [!f] * : *les noms de fichiers ne commençant pas par f*
- *[!0-9] : *les noms de fichiers ne se terminant pas par un chiffre.*

Un raccourci particulier : ~ ("tilde") La plupart des fichiers manipulés au quotidien sont situés dans les répertoires des utilisateurs. Pour faciliter leur manipulation, le shell Unix reconnaît ~ comme raccourci pour indiquer un répertoire utilisateur, qui sera remplacé par le chemin complet du répertoire **avant** l'exécution de la commande.

Ce caractère peut être utilisé seul pour désigner le répertoire d'accueil de l'utilisateur courant ou suivi d'un nom d'utilisateur pour faire référence à un répertoire spécifique.

Pour un utilisateur connecté en tant que *toto*, ces syntaxes sont donc équivalentes :

```
$ ls ~
$ ls ~toto
$ ls /home/toto
```

3.1.10 Les différents types de fichiers

Il existe plusieurs types de fichiers différents dans leur utilisation sous UNIX. Le type associé à un fichier est mémorisé au niveau de son inode, et peut être visualisé à l'aide de la commande ls associé à -l `ls -l`. Ci dessous, chaque type de fichier est suivi du symbole le représentant à l'exécution de cette commande. Le type permet de déterminer quelle action sera appliquée au fichier par le système de gestion des fichiers.

Les fichiers ordinaires (-)

Ces fichiers contiennent une suite de caractères ASCII ou binaire.

Ce sont les fichiers de données des fichiers sources, les fichiers exécutables... qui représente l'essentiel des fichiers manipulés par l'utilisateur. Aucun traitement n'est effectué par le système d'exploitation sur ces fichiers.

> *A noter : les lignes de texte dans UNIX sont terminées par un LF (0x0A (valeur ASCII de LF en hexadecimal)) symbolisé par un \n alors que sous Windows elles sont terminées par un CRLF (0x0D0x0A)*

Contrairement aux systèmes de type Windows, Unix n'accorde aucun traitement spécial aux extensions de fichiers, qui ne sont donc pas obligatoires, mais facilitent le travail de certains outils et l'identifications par l'utilisateur. En voici une liste non exhaustive, purement indicative :

Pour les langages :

.h header langage C

.c fichier source C

.cc fichier source C++ (alternatives : .cpp, .cxx)

.a librairie statique

.sh script shell Bash

| .pl] script Perl

.o binaire objet

Vous pouvez bien sûr tout à fait soumettre à un compilateur C un fichier source nommé "lemeilleurcquejaijamaisecrit" !

Pour la documentation :

.ps Postscript

.tex Tex, LaTex

.eps postscript encapsulé

.html HyperTextMarkLanguage

Autres :

.zip fichier compressé avec l'utilitaire zip

.gz fichier compressé avec l'utilitaire gzip

.bz2 fichier compressé avec l'utilitaire bzip2

.tar fichier compressé avec l'utilitaire tar

Les répertoires (d)

un ensemble de fichiers et / ou de répertoires , composé de la liste de ses composants

Un répertoire peut se lire comme un fichier ordinaire mais se modifie à l'aide de commandes spécifiques

La structure d'un répertoire est un fichier de

- n x 16 octets (n = nombre de fichiers du répertoire) en system V
- n x 512 octets en BSD

Les fichiers liens (l)

Il est possible d'avoir plusieurs noms dans différents répertoires pour un même fichier : ces noms sont des identificateurs du fichier et ont le même numéro d'inode.

Cette relation particulière entre un numéro d'inode et un identificateur s'appelle un *lien*.

Tout fichier possède au minimum un lien. Lorsque le lien est établi dans le même système de fichiers, le lien est dit simple ; les droits du lien sont les mêmes que les droits du fichier puisqu'il s'agit du même objet. Par contre, un lien symbolique peut être sis dans un autre système de fichiers que le fichier d'origine : c'est un pointeur vers l'autre fichier, ses permissions seront indépendantes du fichier source.

Les fichiers spéciaux (b,c)

Ils permettent d'associer un nom de fichier à un périphérique ou à un mécanisme de communication.

Par exemple
– le clavier est un fichier en entrée,
– l'écran est un fichier en sortie,
– l'imprimante est un fichier en sortie
Ils sont stockés dans le répertoire `/dev`, possèdent un inode mais pas de bloc de données et sont gérés par des interfaces d'accès à la ressource physique différents selon leur type (cf figure 3.3) :

les fichiers bloc (type b)

les fichiers caractères (type c) [interface appelé raw, fichiers appelés par extension de langage raw devices]

Certains périphériques possèdent les deux types d'interface (disques par exemple)

Interface mode bloc

Les données transitent par un buffer cache géré par le noyau, ce qui permet d'améliorer les performances en réduisant les accès physiques : lecture anticipée et écriture différée de blocs

Cet interface est principalement utilisé pour la gestion des disques.

Interface mode caractère

Les données sont lues ou écrites directement sans transiter par le cache du système.

Cette interface peut être utilisée pour les diques affectés à un système de gestion de bases de données qui gère son propre mécanisme de buffer cache.

Pour chaque type de périphérique il existe un pilote de périphérique (encore appelé *driver*) qui est une interface logicielle permettant la communication avec ce périphérique (entrées/ sorties entre autres)

Lorsqu'on liste un fichier en mode caractère, on voit que la taille est remplacée par un couple *majeur, mineur*

le majeur correspond à la classe du périphérique utilisée , c'est à dire à un driver parmi tous les drivers gérant ce type d'interface (bloc ou caractère)

le mineur correspond au rang du périphérique dans cette classe

Exemple : lister 3 imprimantes sous /dev (détail de la commande ls dans la partie accès de ce chapitre)

```
$ ls -l /dev
crw--w---- 1 root lp1 6,1 .....
crw--w---- 1 root lp2 6,2 .....
crw--w---- 1 root lp3 6,3 .....
```

"**6**" représente le driver d'imprimante, "1", "2", "3", chaque imprimante

Les pipes nommés (p) Ils sont aussi appelés "fichiers fifos". Ils sont utilisés entre autre par les commandes d'impression et d'ordonnancement de tâches.

Les sockets (s) Ils permettent la communication entre processus et donc entre applications sur la même machine ou à travers le réseau. Ces interfaces sont présentes sous la forme de fichiers .

3.2 Commandes relatives aux fichiers

Nous voyons dans cette leçon une partie des commandes applicables aux fichiers. D'autres seront vues en fin de chapitre 4.

Seules une à deux options (voire aucune) seront développées par commande : rappelez vous que les options peuvent diverger d'un système à l'autre, et qu'il est toujours possible (et conseillé) de consulter la documentation lorsqu'on a un besoin particulier. Les options peuvent, en outre, être complétées lors de nouvelles versions,

le réflexe documentation est donc à privilégier !

3.2.1 Déplacement dans l'arborescence

Nous avons vu qu'il était possible d'accéder à un fichier en le désignant de façon absolue (en indiquant le chemin complet) ou relative (en indiquant seulement le chemin à partir de l'endroit où on se trouve). Pour pouvoir utiliser la seconde notation, il est nécessaire de savoir où on se trouve dans l'arborescence, ce qui, après quelques manipulations est assez difficile (quel est mon répertoire de travail ?)

Pour le connaître, il est possible d'utiliser la commande **pwd** :

Listing 3.2– Affichage du répertoire de travail avec pwd

```
$ pwd
/home/utilisateur/exercices
```

Chaque utilisateur a un "répertoire d'accueil", celui sous lequel il a son espace de travail. (La mise en place du nom de ce répertoire sera vue au chapitre 4) Il est possible, dans la mesure des autorisations qui lui ont été données, de se "promener" à travers l'arborescence du système.

Pour retourner au répertoire d'acceuil de l'utilisateur, il est possible d'utiliser la commande **cd** sans paramètres :

Listing 3.3– Retour au répertoire personnel

```
$ cd
$ pwd
/home/utilisateur
```

Si un répertoire est indiqué en paramètre, sous forme de chemin relatif ou absolu, **cd** se déplace dans l'arborescence et utilise ce répertoire comme nouveau répertoire de travail :

Listing 3.4– Changement de répertoire

```
$ cd /usr/var/www
$ cd exercices
```

La commande **cd** interprète également le caractère "-" de manière spécifique, comme indicateur de retour au répertoire de travail précédent :

Listing 3.5– Retour au répertoire de travail précédent

```
$ pwd
/home/utilisateur
$ cd -
$ pwd
/home/utilisateur/exercices
```

Comme nous l'avons vu dans le début de ce chapitre, les chemins relatifs permettent d'indiquer un chemin par rapport au répertoire de travail, plutôt qu'en indiquant son chemin absolu qui peut être très long. Pour cela, le répertoire de travail sera indiqué par le caractère `.`, et son répertoire parent par `..`. Tous les chemins suivants sont ainsi équivalents, si le répertoire de travail est `/home/utilisateur` :

Listing 3.6– Notations équivalents d'un chemin de fichier

```
$ cd ../utilisateur2
$ cd ./../utilisateur2
$ cd /home/utilisateur2
```

(attention, cette commande peut ne pas être valable sous certains shells)

3.2.2 Création d'un nouveau fichier

La manière la plus simple de créer un nouveau fichier est d'utiliser la commande `touch` : Celle-ci prend en paramètre un chemin de fichier et crée un fichier vide si celui-ci n'existe pas, ou met à jour sa dernière date d'accès le cas échéant

Listing 3.7– Création d'un fichier toto

```
$ touch ./toto
```

La création d'un fichier peut également être réalisée en alimentant le fichier avec la sortie d'une commande. Pour cela, on utilisera le signe > pour créer ou écraser le contenu existant du fichier, et » pour concaténéer le nouveau contenu à la fin des données actuelles. Ceci sera expliqué en détails dans le chapitre suivant dans la section traitant des redirections d'entrée sortie..

Listing 3.8– Exemple d'utilisation de la redirection

```
$ cat > ./toto
ceci est un fichier<enter>
qui contiendra ces 2 lignes de texte<CTRL-D>
```

3.2.3 Visualisation d'un fichier

Le moyen le plus simple d'afficher le contenu d'un fichier est la commande **`cat`** :

Listing 3.9– Affichage d'un fichier

```
$ cat ./toto
ligne1
ligne2
ligne3
```

Si le contenu du fichier ne peut entier dans l'écran de la console, il devient alors intéressant d'utiliser des commandes comme `more` ou `less`, qui permettent de visualiser le contenu d'un texte en paginant par ligne ou par section.

L'accès à la partie du texte non affichée à l'écran s'effectue de la même manière que pour le man en appuyant sur la **touche espace (pour la page suivante)**, sur la **touche <enter> (pour la ligne suivante)**. En bas de chaque page, le caractère : apparaît ; la commande 'h' après ce prompt permet d'accéder aux menus d'aide. La commande "`less`" permet en cours de visualisation d'utiliser les commandes "`vi`" pour se déplacer dans le fichier, ce que ne permet pas la commande "`more`".

Listing 3.10– Affichage d'un fichier page par page

```
$ more ./toto
```

Il existe également deux commandes permettant de ne visualiser que le début ou la fin des fichiers.

La commande

```
head -n nombre fichier
```

affiche les n premières lignes d'un fichier :

Listing 3.11– Affichage du début d'un fichier

```
$ head -n 5 toto
ceci est le texte
a l'interieur de
mon fichier et
je n'affiche que
les 5 premières lignes
```

La commande

```
tail nombre fichier
```

affiche les n dernières lignes du fichier en commançant par le début ou par la fin du fichier :

tail +2 fichier affiche le fichier à partir de la ligne n° 2(début)

tail -5 fichier affiche les 5 dernières lignes en partant de la fin du fichier

La commande **tail** accepte une option très utile : **-f**. Cette option permet de visualiser la fin d'un fichier tout en bénéficiant du rafraîchisement de l'affichage. Ceci est utile pour visualiser un fichier trace ou log lors de l'éxécution d'une procédure posant problème par exemple. La sortie de cette commande se fait par CTRL-C.

Listing 3.12– Affichage avec rafraîchissement de la fin d'un fichier

```
$ tail -f fichier
```

Enfin, la commande `less fichier` est une alternative plus puissante à la commande more, qui permet entre autres d'utiliser les commandes **vi** pour se déplacer dans la sortie affichée.

3.2.4 Suppression d'un fichier

```
rm [-r] nom_fichier
```

permet de supprimer un fichier. L'option **-r** permet de supprimer les descendants de ce fichiers de manière récursive : elle est indispensable pour les répertoires.

Listing 3.13– Suppression de fichiers

```
$ rm /home/utilisateur/exercices/exercice1.txt
$ rm -r /home/utilisateur/exercice
```

3.2.5 Copie d'un fichier ou d'un répertoire

```
cp [-r] source cible
```

permet de copier le fichier indiqué par `source` à l'emplacement indiqué par `destination`. Comme pour la commande rm, elle accepte une option **-r**, utilisée pour les répertoires.

Listing 3.14– Copie de fichier

```
$ cp /home/utilisateur/document1.txt /home/mon_voisin/document1.txt
```

Remarque : Les 2 fichiers sont distincts et sont chacun référencés par leur propre numéro d'inode : il y a duplication des données.

Listing 3.15– Copie de répertoire

```
$ cp -r /home/mesfichiers /home/mon_voisin
```

3.2.6 Création d'un lien vers un fichier

```
ln [-s] nom_du_fichier_actuel deuxième_nom_pour_le_fichier
```

crée un lien (ou alias) vers un fichier :

Listing 3.16– Création d'un lien vers un fichier

```
$ ln /home/utilisateur/premier_fichier /home/utilisateur/fichier
```

L'option **-s** permet de créer des liens entre répertoires ou entre fichiers qui ne sont pas sous le même système de fichiers. On appelle ces liens des liens **symboliques**. Dans ce cas, les "données" du lien contiennent le chemin d'accès et non pas le numéro d'inode du fichier cible.

Remarque : Un lien se supprime par la commande "rm", ce qui signifie que lors de la présence de plusieurs liens pour un fichier, les données ne sont effacées qu'à la commande "rm" sur le dernier lien physique du fichier.

3.2.7 Déplacement d'un fichier

```
mv source cible
```

permet de déplacer un fichier :

Listing 3.17– Déplacement d'un fichier

```
$ mv /home/utilisateur/premier_fichier /home/utilisateur/fichier
```

Remarque : l'ancien chemin vers le fichier est supprimé et remplacé par le nouveau chemin indiqué.

3.2.8 Création et suppression de répertoires

Nous venons de voir une commande "rm" qui permet de supprimer un répertoire avec son contenu. **Si le répertoire est vide**, la suppression se fait à l'aide de la commande `rmdir répertoire` :

Listing 3.18– Suppression d'un répertoire vide

```
rmdir /home/utilisateur/old_exercices
```

A l'inverse, la création d'un répertoire s'effectue avec la commande `mkdir répertoire` :

Listing 3.19– Création d'un répertoire

```
mkdir /home/mesfichiers/deuxieme_rep
```

3.2.9 Lister le contenu d'un répertoire

Une des commandes les plus utilisées est la commande de listage de répertoire, `ls`. Celle-ci comporte de très nombreuses options, utilisées sous cette forme :

```
ls [-options] [chemin]
```

Cette commande balaye la table des inodes.

Voici quelques options de la commande ls :

– Sans option, "ls" donne la liste des fichiers visibles du répertoire

Listing 3.20– Utilisation simple de ls

```
$ ls
monfichier mon2nd_fic fic_temp fichier_debut le_fichier
fic1 fichier_n12 fichier_n2
```

– Avec l'option **-l**, ls inclut également toutes les informations des fichiers (format long) :

Listing 3.21– Utilisation de ls au format long

```
$ ls -l
1 (2)      (3) (4) (5)    (6)     (7)           (8)
-rw-r----- 1 root sys    310 Mar 15 12:13 password.t
-rwxr-xr-x 1 root sys  10650 Jan 30 16:12 execute
drwxrwxrwx 2 bin  bin    320 Jun  9 2000  lib
drwxrwxr-x 7 root sys    144 Mar  8 09:08 etc
```

Nous trouvons tout d'abord 10 caractères accolés

1. le premier caractère indique le type du fichier

   d signifie "directory" , répertoire en français

   - correspond à un fichier ordinaire

   b ou c à un fichier spécial

   l à un lien symbolique

2. les caractères suivants indiquent les permissions du fichier. Il s'agit de 3 fois 3 caractères correspondants :
   – pour les 3 premiers aux permissions du propriétaire du fichier
   – pour les 3 suivants aux permissions pour les membres du groupe
   – pour les 3 derniers aux permissions pour les autres utilisateurs

   Nous verrons le détail de ces permissions dans la leçon suivante.

3. ensuite, vient un chiffre qui représente le nombre de liens du fichier. Pour un répertoire, celui-ci est au minimun à 2 (le répertoire lui-même et son "père"). La ligne du répertoire **etc** contient "**7**" : cela signifie qu'il contient 5 sous-répertoires + lui même + son père (5 + 1 + 1 = 7)

4. vient ensuite le nom du propriétaire
5. puis le nom du groupe auquel ce fichier appartient
6. les chiffres suivants correspondent à la taille du fichier
7. nous trouvons ensuite la date de dernière modification du fichier sous la forme : **mois jour heure** si la modification a eu lieu il y a moins d'un **an, mois jour année** dans le cas contraire (ex pour lib : dernière modification le 9 Juin 2000)
8. et en dernier, le nom du fichier !

La commande ls admet un nombre important d'options : Nous verrons seulement les plus utilisées dans ce cours

ls -l affiche la liste avec le détail pour chaque fichier (vu ci dessus)

ls -i affiche la liste des fichiers avec pour chacun son numéro d'inode

ls -R affiche récursivement le contenu du répertoire et de ses successeurs

ls -a affiche tous les fichiers , y compris les fichiers cachés

ls -lt affiche les fichiers par ordre de dernière modification (du plus récent au plus ancien)

ls -ld dir affiche le détail des permissions pour le répertoire dir

Toutes les options peuvent se combiner entre elles, à la condition qu'elles ne s'excluent pas l'une l'autre.

3.2.10 Concaténer le contenu de plusieurs fichiers

la concaténation se fait avec la commande **cat** que nous avons déjà évoquée précédemment :

Listing 3.22– Concaténation de fichiers

```
$ cat fichier1 fichier2 ... fichiern > fichier_total
```

Il est bien sûr possible d'utiliser l'opérateur de redirection "»" pour concaténer plusieurs fichiers à la suite d'un fichier existant.

3.2.11 Comparaison de fichiers

Deux commandes permettent de comparer des fichiers :

– La commande `diff fich1 fich2`, qui compare le contenu des deux fichiers de type ASCII indiqués. Cette commande ignore les lignes identiqués, et n'affiche que les lignes modifiées.

Listing 3.23– Comparaison de contenu avec diff

```
$ diff fichier1 fichier2
1c1
< premier
---
> deuxième
3c3,4
< et ne continue pas
---
> et s'arrête
> à cette ligne-là
```

– La commande `cmp [-l] [-s] file1 file2` affiche les différences entre deux fichiers quelque soit leur type (ASCII ou binaire)

1. sans option, on obtient juste la 1ère postion où les fichiers diffèrent
2. l'option -s renvoie en code retour seulement 0 si les deux fichiers sont identiques, 1 s'il y a une différence
3. l'option -l permet de constituer un tableau à 3 colonnes avec en colonne 1 la position sur laquelle a éé détectée la différence, sur la deuxième la valeur dans le premier fichier, sur la troisième la valeur dans le troisième fichier. La valeur affichée est la valeur en octal du caractère ASCII

Listing 3.24– Exemples de comparaisons avec cmp

```
$ cmp -s fichier1 fichier2
1
$ cmp -s fichier1 fichier2
fichier1 fichier2 differ: char 5,line 1 le 5ème
caractère est "p" dans le premier, "d" dans le second
$ cmp -l fichier1 fichier2
5 160 144 5ème caractère, 160=p 144=d
6 162 145 6ème caractère, 162=r, 145=e
7 145 165 etc ..
.
8 155 170
.
.
.
67 163 64
68  12 164
```

3.2.12 Division en plusieurs parties d'un fichier

La commande `split -ln fichier` permet de découper fichier en petits fichiers de n lignes chacun (une ligne se terminant par un LF)

Les découpages seront faits dans des fichiers **xaa, xab,xac ..., xah** pour le 8ème et

ainsi de suite. D'autres options permettent de choisir le nombre de lettres de l'extension par exemple (faire man split pour plus d'explications)

Listing 3.25– Découpage d'un fichier

```
$ split -l 150 mon_fichier
```

Cette commande créera des fichiers nommés xaa pour le premier, xab pour le second ... xag pour le 7ème contenant chacun 150 lignes.

3.2.13 Manipulation des chemins de fichiers

```
basename string [suffixe]
```

permet de connaître le nom du fichier sans son répertoire parent

Listing 3.26– Récupération du nom d'un fichier

```
$ basename /home/mes_fichiers/mon_fichier
mon_fichier
```

Pour débarasser le nom du fichier d'un suffixe quelconque, il faut préciser ce suffixe en paramètre (*suffixe au sens "derniers caractères du mot")* :

Listing 3.27– Récupération du nom de fichier sans extension

```
$ basename /home/mes_fichiers/mon_fichier.ps .ps
mon_fichier
```

Remarque : Cette commande est principalement utilisée à l'intérieur de scripts, lorsqu'on parcourt un catalogue afin d'utiliser dans une commande suivante le nom du fichier entier ou le début de ce nom.

```
dirname string
```

permet de connaître le chemin d'accès du fichier

Listing 3.28– Récupération du répertoire parent d'un fichier

```
$ dirname /home/mes_fichiers/mon_fichier
/home/mes_fichiers
```

cette commande est elle aussi principalement utilisée à l'intérieur de scripts.

3.2.14 Récupération d'informations sur des fichiers

La commande `file fichier` permet de connaître la nature du fichier cette information est stockée sous forme d'un nombre dans les 2 premiers octets du fichiers, et la correspondance entre ce nombre et un libellé en clair est établie grâce au contenu du fichier **magic**. Le résultat de la commande file dépendra du contenu (plus ou moins complet selon les machines) du fichier magic...

Listing 3.29– Détection d'informations sur le contenu d'un fichier

```
$ file /home/mes_fichiers/mon_fichier
/home/mes_fichiers/mon_fichier: ascii text
$ file /home/mes_fichiers
/home/mes_fichiers: directory
```

La commande **`type`** `fichier` permet de connaître le type et le chemin d'accès d'une commande :

Listing 3.30– Récupération d'information sur une commande

```
$ type ls
ls is /usr/bin/ls
$ type rm
rm is a fonction
rm()
{
/bin/rm
*
}
$ type pwd
pwd is a shell builtin
```

3.3 Droits d'accès à un fichier

3.3.1 Les permissions

Nous verrons au chapitre 4 comment se gèrent les utilisateurs. D'ores et déjà, il faut retenir qu'**un utilisateur appartient à un groupe d'utilisateurs qui ont des droits sur les fichiers**. Nous verrons ici uniquement les permissions de base et les commandes permettant de modifier ces permissions.

Chaque fichier UNIX possède 3 types de permissions, que nous visualisons à l'aide de la commande ls -l

1. permission de lire "r"
2. permission d'écrire "w"
3. permission d'exécuter "x"

UNIX distingue 3 catégories d'utilisateurs

- le propriétaire du fichier
- les membres de son groupe
- les autres

Pour chacune de ces catégories, chacune des 3 permissions peut être accordée ou refusée, ce qui donne pour chaque fichier 9 permissions.

Ces permissions sont données ou révoquées **par le propriétaire** du fichier, et un utilisateur, le "super-utilisateur" qui a accès à tous les fichiers du système.

La matérialiation d'une permission non accordée est le tiret "**-**".

Dans le schéma ci dessous, on voit que seul le propriétaire a le droit de modifer my_file.

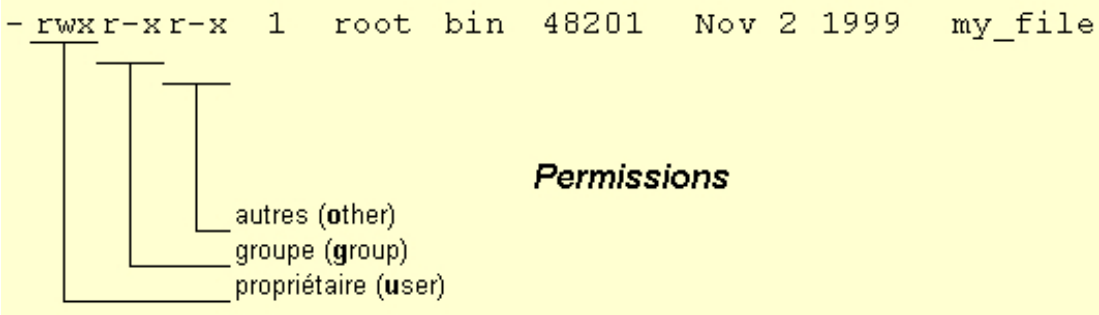

FIGURE 3.4 – Les permissions sur les fichiers

Les permissions accordées aux fichiers

r permission de lire le contenu du fichier (Read)

w permission de modifier le contenu du fichier (Write)

x permission de demander au noyau ou à l'interpréteur de commande d'exécuter le fichier (eXecute)

Les permissions accordées aux répertoires

r permission de lire le contenu du répertoire, commande "ls" possible (Read)

w permission de modifier le contenu du répertoire, possibilité de création ou suppression d'un fichier dans le répertoire (Write)

x permission d'utiliser ce répertoire pour accéder à un de ses fichiers, commande "ls -l" possible (eXplore)

Exemple :

```
drwxr----- 1 albert etudes 48201 Nov 2 1999 My_Rep
```

ce répertoire peut être listé avec la commande ls -l uniquement par son propriétaire (albert) et visualisé avec ls par les membres du groupe (etudes). Tous les autres n'ont aucun accès autorisé à ce répertoire. Les membres du groupe "etudes" ne peuvent pas faire la commande ls -l (pas d'eXploration sur le groupe).

3.3.2 Modification du propriétaire d'un fichier

Dans toutes les commandes qui suivent, l'option **-R** indique qu'il faut appliquer récursivement la modification sur l'arborescence d'un répertoire

La modification se fait à l'aide de la commande **chown** (change owner)

Attention, le propriétaire d'un fichier peut seulement changer le groupe d'appartenance de ce fichier à condition qu'il soit aussi dans ce groupe. En d'autres termes, un utilisateur ne peut pas se déposséder d'un fichier. Seul le "super utilisateur" peut le faire.

```
chown [-R] propriétaire[:group] [fichier ou répertoire]
```

Exemple :

```
drwx------ 1 albert etudes 48201 Nov 2 1999 My_Rep
```

ce répertoire appartient à "albert" du groupe "etudes"

```
chown -R gilles My_Rep
```

```
drwx------ 1 gilles etudes 48201 Nov 2 1999 My_Rep
```

```
chown -R virginie:production My_Rep
```

```
drwx------ 1 virginie production 48201 Nov 2 1999 My_Rep
```

3.3.3 Modifications des permissions

La modification se fait à l'aide de la commande **chmod** (change mode) qui accepte 2 syntaxes

1. chmod [-R] nnn [fichier ou répertoire]
2. chmod [-R] [X] opérateur Y [fichier ou répertoire]

```
chmod [-R] nnn [fichier ou répertoire]
```

Dans la première syntaxe, **nnn** est la représentation en octal de la permission. Chacun des 9 chiffres binaires correspondent aux permissions : il est positionné à 1 si la permission est donnée, à 0 sinon.

Les permissions pour le propriétaire sont codées sur 3 bits, ce qui représente au total 24 possibilités, soit une valeur de 0 à 7, 0 représentant tous les bits à 0, 7 tous à 1 . (Dans les faits, on a 23 possibilités car 000 n'a pas de sens !).

Il en va de même pour le groupe et "les autres".

exemple :

```
-rwx------ 1 albert etudes 48201 Nov 2 1999 My_Prog
```

```
chmod 751 My_Prog
```

La valeur en octal **751** correspond en binaire à **111 101 001**. Les nouvelles permissions sont :

```
-rwxr-x--x 1 albert etudes 48201 Nov 2 1999 My_Prog
```

```
chmod [-R] [X] opérateur Y [fichier ou répertoire]
```

Dans cette seconde syntaxe, on indique explicitement la classe concernée et le droit préfixé par un opérateur.

X correspond à la catégorie concernée :

u propriétaire (user)

g groupe (group)

o other (autres)

a all (tous : option par défaut)

opérateur correspond à la modification du droit

+ ajout

- retrait

= affectation

Y correspond au droit ou à la catégorie où prendre les droits

r lire

w écrire

x exécuter

exemple :

```
-rwxrw-rw-1 albert etudes 48201 Nov 2 1999 My_Prog
```

```
chmod +x My_Prog
```

My_Prog peut être exécuté par tout le monde

```
-rwxrwxrwx 1 albert etudes 48201 Nov 2 1999 My_Prog
```

```
chmod o-w My_Prog
```

My_Prog ne peut plus être modifié par "les autres"

```
-rwxrwxr-x 1 albert etudes 48201 Nov 2 1999 My_Prog
```

```
chmod g=o My_Prog
```

Les membres de mon groupe obtiennent sur My_Prog les mêmes droits que "les autres"

```
-rwxr-xr-x 1 albert etudes 48201 Nov 2 1999 My_Prog
```

3.3.4 Les permissions par défaut

Lors de la création d'un fichier, le noyau se base sur une valeur contenue dans une variable **umask**. Cette variable est positionnée pour le système mais peut être modifiée et personnalisée pour chaque utilisateur (nous verrons cela dans l'environnement utilisateur)

"**umask**" permet de préciser au système les permissions qu'il doit affecter par défaut à la création d'un nouveau répertoire ou d'un nouveau fichier.

La commande `umask [masque]` permet sans paramètre de visualiser la valeur actuelle de umask, avec paramètre de modifier cette valeur

Les permissions sont codées sur 12 bits, dont 9 correspondent aux droits en lecture/écriture/exécution sur le fichier. Nous nous intéressons à ces derniers.

les 3 premiers bits correspond aux permissions de l'utilisateur

les 3 bits suivants aux permissions du groupe

les 3 bits suivants aux permissions de tout le monde

umask est aussi codé sur 3 octets

Que fait le système face à une création de répertoire ? Il donne en permissions le complément à "1" de umask

Exemple :

```
umask 050
```

0 en octal pour l'utilisateur soit en binaire 000

5 en octal pour le groupe soit en bianire 101

0 en octal pour les autres soit en bianaire 000

Le complément à 1 de `000 101 000` est `111 010 111` (`rwx-w-rwx`), ce qui "traduit" en octal donne **727** .

La création d'un répertoire avec umask à **050** donnera à ce répertoire les permissions

```
rwx-w-rwx
```

Et pour une création de fichier ? Le système commence son calcul comme pour un répertoire, et pour mon umask à 050 j'arrive à nouveau à

`111 010 111` (`rwx-w-rwx`)

mais ensuite :

il supprime les permissions "exécution" à "1" ce qui donne

`110 010 110` (`rw-w-rw-`) soit en octal **626**

Autre exemple :

`umask 542` (101 100 010)
– répertoire créé avec 010 011 101 (complément à 1) -w–wxr-x
– fichier créé avec 010 010 100 (suppression des "exécutions" à 1) -w–w-r–
Vous constatez que les exemples ci dessus sont somme toute très farfelus ! Ils ne servent qu'à démontrer le mécanisme sous jacent . Voici des exemples plus conformes à la réalité :

`umask` `0`

le fichier est créé avec les permissions "**666**"

```
-rw-rw-rw-1 albert etudes 48201 Nov 2 1999 My_Prog
```

`umask` `777`

le fichier est créé sans aucune permission !

```
----------1 albert etudes 48201 Nov 2 1999 My_Prog
```

`umask` `222` le fichier est créé avec les permissions "**444**"

```
-r--r--r--1 albert etudes 48201 Nov 2 1999 My_Prog
```

`umask` `066` le fichier est créé avec les permissions "**600**"

```
-rw-------1 albert etudes 48201 Nov 2 1999 My_Prog
```

umask accepte aussi une syntaxe reprenant celle de chmod (syntaxe 2) ainsi **`umask`** `u=rwx,g=x,o=x` correspond à**`umask`** `066`

3.3.5 Les permissions spéciales

Au gré de vos commandes "ls -l", vous verrez des permissions différentes, comme "**s**"

```
-rwsrw-rw-1 albert etudes 48201 Nov 2 1999 My_Prog
```

Nous aborderons la signification de ces permissions dans le chapitre 6, mais nous voyons d'ores et déjà comment les positionner. Ces permissions sont codées sur le 4ème octet, selon les mêmes règles que précédemment. Elles ne sont pas directement visualisables, et se surajoutent à l'affichage avec le bit "x".

Pour positionner la valeur "s" par exemple, la syntaxe est `chmod 4755 My_Prog` ou `chmod u=rwxs,go=rx My_Prog`

```
-rwsr-xr-x 1 albert etudes 48201 Nov 2 1999 My_Prog
```

Cette valeur "s" peut aussi se positionner au niveau du groupe et la syntaxe devient alors `chmod 2755 My_Prog` ou `chmod u=rwx,g=rxs,o=rx`

```
-rwxr-sr-x 1 albert etudes 48201 Nov 2 1999 My_Prog
```

Vous trouverez aussi parfois une permission "**t**" au niveau "other", qui se positionne de la façon suivante `chmod 1755 My_Prog` ou `chmod u=rwx,g=rx,o=rxt My_Prog`

```
-rwxr-xr-t 1 albert etudes 48201 Nov 2 1999 My_Prog
```

3.4 Exercices

Un seul exercice dans ce chapitre, qui consiste à revoir et pratiquer les commandes vues

Exercice 1 :

Entrez les commandes vous permettant de répondre aux questions suivantes

Question 1.1 : où êtes vous dans la hierarchie ?

Question 1.2 : y a t-il des fichiers, des répertoires ?

Question 1.3 : entrez du texte dans un fichier Mon_fichier

Question 1.4 : listez le contenu de Mon_fichier

Question 1.5 : listez votre catalogue

Question 1.6 : listez les catalogues /bin /dev

Question 1.7 : créez sous votre catalogue deux répertoires : source et data

Question 1.8 : positionnez vous sous source

Question 1.9 : listez votre répertoire

Question 1.10 : revenez sous votre répertoire de départ et détruisez "source"

Question 1.11 : créer un deuxième fichier Mon_fichier_2

Question 1.12 : copiez chaque fichier en nom de fichier.old

Question 1.13 : créer un répertoire old

Question 1.14 : déplacez les fichiers avec l'extension old sous le répertoire old

Question 1.15 : copiez les fichiers sans extension du répertoire courant sous data sous votre répertoire de départ, créez un lien Mon_lien équivalent à Mon_fichier_2

Question 1.16 : listez les deux fichiers Mon_lien et Mon_fichier_2 en affichant leur numéro d'inode : que remarquez vous ?

Question 1.17 : notez les permissions de Mon_lien et Mon_fichier_2 ; changez les permissions de Mon_lien et regardez à nouveau les permissions de Mon_lien et Mon_fichier_2 : que remarquez vous ? pourquoi ?

Question 1.18 : supprimez Mon_lien . Mon_fichier_2 a t il disparu ?

Question 1.19 : notez le numéro d'inode de Mon_fichier_2 . Faites : mv Mon_fichier_2 Mon_fichier_3. Quel est le numéro d'inode de Mon_fichier_3 ?

Question 1.20 : effacez tous les fichiers crées

3.5 Correction des exercices

Correction Exercice 1 :

Solution question 1.1 : où êtes vous dans la hierarchie ?

```
pwd
```

Solution question 1.2 : y a t-il des fichiers, des répertoires ?

```
ls -l
```

Solution question 1.3 : entrez du texte dans un fichier Mon_fichier

```
cat > Mon_fichier ceci est un texte au hasard vous avez inscrit
 ce que vous souhaitez CTRL-D
```

Solution question 1.4 : listez le contenu de Mon_fichier

```
more Mon_fichier ou pg Mon_fichier
```

Solution question 1.5 : listez votre catalogue

```
ls -l
```

Solution question 1.6 : listez les catalogues /bin /dev

```
ls -l /devls /bin
```

Solution question 1.7 : créez sous votre catalogue deux répertoires : source et data

```
mkdir source mkdir data
```

Solution question 1.8 : positionnez vous sous source

`cd source` ou `cd ./source`

Solution question 1.9 : listez votre répertoire

```
ls -l
```

Solution question 1.10 : revenez sous votre répertoire de départ et détruisez "source"

```
cd .. rmdir source
```

Solution question 1.11 : créer un deuxième fichier Mon_fichier_2

```
cat > Mon_fichier_2 à nouveau du texte CTRL-D
```

Solution question 1.12 : copiez chaque fichier en nom de fichier.old

```
cp Mon_fichier Mon_fichier.old cp Mon_fichier_2 Mon_ficher_2.old
```

Solution question 1.13 : créer un répertoire old

```
mkdir old
```

Solution question 1.14 : déplacez les fichiers avec l'extension old sous le répertoire old

`mv Mon_fichier.old ./old/Mon_fichier.old` ou `mv Mon_fichier_2.old old/Mon_fichier_2.old` (syntaxe old ou ./old valides dans les deux cas)

Solution question 1.15 : copiez les fichiers sans extension sous le répertoire data

```
cp Mon_fichier data/Mon_fichier;cp Mon_fichier_2 ./data/Mon_fichier_2
```

Solution question 1.16 : sous votre répertoire de départ, créez un lien Mon_lien équivalent à Mon_fichier_2

```
ln Mon_fichier_2 Mon_lien
```

Solution question 1.17 : listez les deux fichiers Mon_lien et Mon_fichier_2 en affichant leur numéro d'inode : que remarquez vous ?

ls -i : ils ont le même numéro d'inode,et Mon_fichier_2 a deux liens

Solution question 1.18 : notez les permissions de Mon_lien et Mon_fichier_2 ; changez les permissions de Mon_lien et regardez à nouveau les permissions de Mon_lien et Mon_fichier_2 : que remarquez vous ? pourquoi ?

ls -l : ils ont toujours les mêmes permissions car il s'agit du même fichier.

Solution question 1.19 : supprimez Mon_lien . Mon_fichier_2 a t il disparu ? Listez en détail le catalogue, que remarquez vous ?

```
rm Mon_lien
```

Mon_fichier_2 existe toujours, il n'a plus qu'un lien.

Solution question 1.20 : notez le numéro d'inode de Mon_fichier_2 . Faites : mv Mon_fichier_2 Mon_fichier_3. Quel est le numéro d'inode de Mon_fichier_3 ?

Le numéro d'inode ne change pas, c'est le nom du fichier qui a changé.

Solution question 1.21 : effacez tous les fichiers

en fonction du nom des fichiers trouvés, rm fic fic1 ..

Chapitre 4

Le shell et les environnements - BH -

Dans ce troisième chapitre, nous présentons les interpréteurs de commandes. Ces interpréteurs, véritables langages de commandes permettent entre autre d'automatiser des traitements.

4.1 Les interpréteurs de commandes ou shell

Le Shell est l'interface entre le système d'exploitation et l'utilisateur. Plus concrétement, le shell lit une ligne de commande saisie par l'utilisateur, interprète sa signification, exécute la commande (en la soumettant au système d'exploitation) et retourne le résulat sur les sorties.

Nous verrons par la suite (chapitre gestion des utilisateurs) que chaque utilisateur possède un shell par défaut précisé dans le fichier `/etc/passwd`.

Il existe plusieurs type de shell, d'où le pluriel employé pour cette leçon.

Les Shell sont des langages interprétés dans lesquels on peut utiliser et décrire des variables. Des opérateurs de test (`if - else`) ainsi que des opérateurs de boucle (`for - while`) permettent la réalisation de fichiers de commandes. Il est également possible d'enchaîner des commandes à l'aide de `tubes` ou `pipes` ainsi que de redirectionner les entrées-sorties des commandes.

4.1.1 Les différents shells :

Bourne Shell : " sh " ou " bsh "

Il est le shell de la version 7 d'UNIX (le plus ancien des shell présenté). Il reste par défaut le shell pour la configuration des systèmes et se trouve sous /bin/sh sur la plupart des versions UNIX.

C Shell : "csh"

C'est le plus proche du C par sa syntaxe. Il permet de gérer des variables tableaux et des variables de type numérique. Tous les opérateurs du langage C sont disponibles dans le C Shell . Il est de moins en moins utilisé. Les procédures `csh` ne s'exécutent pas sous `sh` et sous `ksh`.

Korn shell : "ksh"

Il est compatible avec le Bourne shell et inclut des fonctionnalités du C Shell.

Il permet la définition de variables tableaux, numériques. Tous les opérateurs du langage C sont compris. Il peut gèrer l'historique des commandes au niveau invite de commande et permet la définition de fonctions. Plusieurs éditeurs de texte sont intégrés dans ce shell dont Vi. Sous `ksh`, on peut personnaliser le `prompt` en incorporant par exemple le nom de la machine, le répertoire courant, etc.

Bourne Again shell : "bash"

C'est un shell développé sous licence "libre" (projet GNU). Il est basé sur le Bourne shell et intègre des fonctionnalités de `ksh` et `csh`. C'est le shell par défaut de nombreux UNIX libres et de Mac OS X.

En résumé

	CSH	KSH	BSH	BASH
Historique des commandes	Non	Oui	Oui	oui
Editeur de ligne	Non	Oui	Non	oui
Syntaxe de boucles	While ... end	While ... do done	While ... do done	While ... do done
Goto	Oui	Non	Non	Non

4.1.2 La soumission de commandes

Lorsque vous avez ouvert une session et que vous soumettez une commande, la soumission de ce que vous venez d'entrer au clavier se fait en appuyant sur la touche <enter> qui a pour effet d'envoyer le caractère "fin de ligne" au shell.

Le shell interprète ce caractère comme la fin de la commande et l'ordre d'exécuter ce qui précède. Un autre caractère est interprété de la même façon, c'est le caractère ";".

Listing 4.1– La soumission de commandes

```
@ cd /chezmoi <enter>
@ bash  monFichierDeCommande <enter>
```

donne le même résultat que :

Listing 4.2– La soumission de commandes sur une seule ligne

```
@ cd /chezmoi;bash monFichierDeCommande <enter>
```

Le shell exécutera d'abord la commande cd /chezmoi puis la commande bash mon_fichier_de_commande

A l'intérieur d'un script, vous pouvez ainsi mettre plusieurs commandes à la suite l'une de l'autre, chaque " ;" sera interprété comme soumission de la commande précédente.

A l'inverse, vous allez sans doute avoir des lignes très longues, où, pour des facilités de lecture vous souhaiteriez pouvoir lire votre ligne sans avoir à tabuler. Comment aller à la ligne sans que le shell interprète ce caractère ? Le caractère \ placé devant le caractère "fin de ligne" empêche son interprétation par le shell.

Listing 4.3– La soumission d'une commande sur plusieurs lignes

```
@ ls \<enter>
> fich<enter>
```

la 1ère ligne indique : après les caractères "ls" je veux faire "enter" sur mon clavier
la seconde : le shell m'affiche le prompt > pour me signifier qu'il attend la suite (par exemple, `fich`)

4.1.3 Les différents types de commandes

Les shells exécutent deux types de commandes :

Les commandes internes

Elles utilisent une fonction du shell, elles font partie du code du shell. Ce sont des mots réservés du langage shell. Ces commandes sont implantées dans le shell pour plusieurs raisons : pour avoir de meilleures performances (ex : `kill`), car elles doivent pouvoir modifier l'envirronnnement du shell courant (ex : `cd`).

Il n'y a pas création de processus fils pour les exécuter : c'est le processus shell qui exécute la commande interne. Donc, les commandes internes peuvent ***modifier*** l'environnement du processus shell courant.

La liste des commandes internes est donnée dans les pages mannuelles du `bash` : `man bash` puis rubrique `shell builtin commands`.

Liste des commandes internes :

., alias, bg, bind, break, builtin, bye, case, cd, command, continue, declare, dirs, echo, enable, eval, exec, exit, export, fc, fg, for, getopts, hash, help, history, if, jobs, kill, let, local, logout, popd, pushd, pwd, read, readonly, return, set, shift, source, suspend, test, times, trap, type, typeset, ulimit, umask, unalias, unset, until, wait, while

Listing 4.4– Commande `type`

```
@ type alias
alias is a shell builtin
```

*La commande **type** donne le type et le chemin d'accès d'une commande.*

Les commandes externes

Elles créent un processus de shell et exécutent la commande dans ce shell fils. Ces commandes ***ne modifient donc pas*** l'environnement du processus shell courant. Nous reviendrons sur ces notions dans le chapitre 5 consacré aux processus.

Listing 4.5– Commande `type`

```
@ type cp
cp is /bin/cp
```

On ne peut pas faire une liste exhaustive des commandes externes, tout programme que vous créez est une commande externe.

Bash : le shell présenté dans ce cours

Dans la suite de ce chapitre nous présentons les principes généraux du **bash shell**. Le bash shell est un sujet suffisament vaste pour nécessiter à lui seul un cours. A l'issu de ce chapitre, vous aurez néanmoins la possibilité d'écrire et surtout de comprendre les shells que vous rencontrerez.

4.1.4 La redirection des entrées/sorties

Lorsque l'exécution d'une commande (appelée processus, cf chapitre 5) est lancée, une table des descripteurs de fichiers ouverts par ce processus est créée. Cette table contient une entrée par fichier manipulé par la commande (ou processus). Dans cette

table, les fichiers sont identifiés par leur index dans la table : un numéro appelé **descripteur de fichier** ou **file descriptor** ou fd. Les trois premières entrées de cette table (index 0, 1 et 2) sont automatiquement initialisées par trois fichiers (systématiquement ouverts à chaque lancement d'un processus) :

- le descripteur de fichiers **0** : correspond à l'entrée standard du processus ou **stdin**, il est initialisé avec le fichier spécial correspondant au clavier.
- le descripteur de fichiers **1** : correspond à la sortie standard (l'écran) du processus ou **stdout**, il est initialisé avec le fichier spécial correspondant à l'écran.
- le descripteur de fichiers **2** : correspond à la sortie des erreurs ou **stderr** du processus, il est initialisé avec le fichier spécial correspondant à l'écran.

On peut modifier l'initialisation automatique des trois premières entrées de la table des fichiers ouverts d'un processus en redirigeant les entrées/sorties de ce processus sur des fichiers.

La redirection des données en sortie

Nous avons vu au chapitre 2 que l'on pouvait créer un fichier avec le résultat d'une commande.

Listing 4.6– Redirection de sortie

```
@ cat monFichier > ./uneCopieDeMonFichier
```

Lors de cette manipulation, le résultat de la commande n'est pas affiché à l'écran mais copié dans le fichier spécifié (`uneCopieDeMonFichier`). Dans ce cas, la sortie standard du processus (ayant pour descripteur de fichier 1) est redirigée sur le fichier `uneCopieDeMonFichier`.

Listing 4.7– Redirection de sortie

```
@ cat monFichier 1> ./uneCopieDeMonFichier
```

Dans cette commande on nomme explicitement la sortie standard (`fd=1`). Nous pouvons bien sûr remplacer > par >>

Listing 4.8– Redirection de sortie et erreur

```
@ commandex > fichierResultat 2>fichierDesErreurs
```

Les erreurs de la commande `commandex` sont redirigées dans un fichier et le résultat correct dans un autre.

Listing 4.9– Redirection de sortie et erreur dans un même fichier

```
@ commandex > fic 2>&1 (&1 représente le file descriptor 1)
```

Les erreurs sont redirigées dans le même fichier que le résultat de la commande.

A noter l'existence d'un fichier "poubelle" dans lequel il est possible de rediriger les erreurs, dans le cas de procédures génératrices d'erreurs connues et sans intérêt. Ce fichier "poubelle" est `/dev/null`.

Listing 4.10– Redirection des erreurs dans le fichier poubelle

```
@ commandey 1> fichierRes 2>/dev/null
```

La redirection des données en entrée

Cette redirection permet de passer des informations à une commande en entrée.

Listing 4.11– Redirection des entrées d'une commande

```
@ more < ./fic
```

donne le résultat de la commande `more` appliqué au contenu du fichier `./fic` ce qui revient à lister le fichier `./fic` !

Il est possible de rediriger une commande en entrée pour simuler l'entrée clavier. Le shell lira les lignes qu'il considère comme étant STDIN jusqu'à la rencontre de la chaîne de caractères indiquée à la suite de "<<" . Cette procédure pourra être utilisée dans les fichiers de commandes que nous voyons à la suite.

Listing 4.12– Création d'un fichier

```
@ cat > mon_fichier <<FIN
ceci est le remplissage de mon fichier
et s'arrêtera
à la rencontre de l'entree
FIN
```

Le contenu de `monFichier` est :

```
@ more monFichier
ceci est le remplissage
de mon fichier
et s'arrêtera
à la rencontre de l'entree
```

4.2 Les fichiers de commandes

Il est possible de créer un fichier contenant une suite de commandes que le shell pourra exécuter. On parlera alors de fichiers de commandes, shell-script ou procédures.

4.2.1 Exécution de fichiers de commandes

Il y a différentes manières d'exécuter un shell-script :

Comme une commande :

Un fichier de commande peut être déclarer comme une commande binaire (comme par exemple `cp`). Il suffit pour cela de lui attribuer des droits d'exécution (commande `chmod`) puis de lancer la procédure comme si c'était une commande classique.

Listing 4.13– Exécution d'un script comme une commande

```
@ chmod +x monFichierDeCommande
@ monFichierDeCommande
```

La deuxième ligne exécute toutes les commandes contenues dans `monFichierDeCommande`.

Au moment de l'exécution de la commande, le système analyse l'entête du fichier, si elle ne décrit pas un fichier binaire, il interprète le fichier comme étant un fichier de commandes, il lance donc un processus shell pour l'interpréter.

Il faut que le chemin d'accès au fichier de commandes soit dans la variable d'environnement `PATH` (si le fichier de commandes est dans le répertoire courant, la variable `PATH` doit contenir le point).

Dans un autre shell

Il y a création d'un nouveau shell qui interprète les commandes du fichier de commandes.

La commande `bash` prend en paramètre un nom de fichier de commandes qui est exécuté par un nouveau shell (ses entrées sont redirigées sur le fichier). Sinon, s'il n'y a pas de paramètres, elle crée simplement un nouveau shell qui lit ses entrées sur l'entrée standard.

Listing 4.14– Exécution d'un script dans un autre shell

```
@ bash monFichierDeCommande
```

Dans ce cas, le ficher `monFichierDeCommande` n'a pas besoin d'avoir des droits d'exécution

Dans le processus courant

On peut exécuter un fichier de commandes en utilisant la commande interne "**.**" qui permet de lire et d'exécuter les commandes se trouvant dans le fichier de commandes passé en paramètre sans créer de nouveaux processus.

Comme "." est une commande, il faut un espace avant le nom du fichier de commandes à exécuter.

Il n'y a pas de nouveau processus shell créé pour exécuter le fichier de commandes.

Listing 4.15– Exécution d'un script dans le processus courant

```
@ . monFichierDeCommande
```

Il est fortement conseillé de revenir sur ces notions après avoir travaillé le chapitre sur les processus.

4.2.2 Les commentaires

Les lignes de commentaires se signalent par un # en 1ère colonne d'une ligne d'un fichier de commandes.. On trouve une ligne de commentaires un peu particulière au début d'un fichier de commandes `shell`, il s'agit par exemple de :

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

Lorsque le fichier de commande est soumis, le shell lit quel est le type de ce fichier La lecture de cette première ligne avec le commentaire **#** suivi du point d'exclamation **!** lui permet de connaître le nom de l'exécutable à qui soumettre ce fichier de commande. La commande `file` appliquée à un fichier donne le type de ce fichier : elle utilise les mêmes informations pour donner son résultat.

Par exemple, si la première ligne d'un fichier de commandes est :

```
#! /chezmoi/mon_interpreteur
```

Le shell soumettra les lignes qui suivent à votre interpréteur de lignes de commandes !

4.2.3 La mise au point

En phase de mise au point, il est intéressant de pouvoir suivre le déroulement de l'exécution d'un fichier de commandes et donc de comprendre où se trouve l'éventuel problème.

Pour se faire, l'exécution se lance en rajoutant **-v** ou **-x** à la commande shell explicite. Ces options permettent de visualiser le code pendant qu'il s'exécute. Une autre option peut s'avérer utile dans le cas de scripts particulièrement longs : l'option **-e** . Dans ce cas, l'exécution s'arrêtera à la première erreur rencontrée.

En résumé :

-v liste la ligne telle qu'elle a été codée

-x liste la ligne après résolution des variables s'il y en a sur cette ligne

-e arrête la procédure à la première erreur

Listing 4.16– Mise au point

```
@ bash –v monFichierDeCommande
```

Ces options sont ici positionnées lors du lancement du fichier de commandes, il est possible de les positionner de façon permanente pour le shell.

4.2.4 Sortie de fichier de commandes

La sortie prématurée d'un fichier de commandes se fait par la commande **exit** [n]. "n" est le code retour ou status de la procédure. Nous verrons par la suite qu'il est possible de le récupérer.

Listing 4.17– Sortie de fichier de commandes

```
...
exit 1
```

Le code de retour du fichier de commandes sera 1

4.2.5 Deux commandes utiles

La commande which

Les commandes shell comme les programmes compilés peuvent avoir des homonymes sur votre système. La commande **which** vous permet de connaître le nom complet du fichier que vous voulez utiliser.

Listing 4.18– La commande `which`

```
@ which more
/bin/more
```

cela signifie que vous utilisez la commande `more` stockée dans le fichier suivant :`/bin/more`. Vous verrez plus loin dans ce chapitre en cas d'homonymes comment sera choisi le fichier exécuté.

La commande whereis

Elle vous permet de connaître le(s) emplacements des exécutables des sources et de la documentation sur une commande.

Listing 4.19– La commande `whereis`

```
@ whereis more
more: /bin/more /usr/share/man/man1/more.1.gz
```

cela signifie que l'exécutable est `/bin/more` et que sa documentation est dans le fichier `/usr/share/man/man1/more.1.gz`

4.3 Les structures de contrôle

Le shell est un LANGAGE interprété. Comme tout langage de programmation il comprend des opérateurs de test et d'itération. Il admet une syntaxe pour laquelle certains caractères ont une signification particulière.

4.3.1 Les commandes echo et read

Ces commandes sont utilisées dans les prochains exemples du cours.

La commande **echo** envoie le texte qui suit sur la sortie standard (l'écran en général).

Listing 4.20– La commande `echo`

```
@ echo Bonjour
Bonjour
```

affiche Bonjour sur l'écran

A l'inverse, la commande **read** permet de lire l'entrée standard :

Listing 4.21– La commande `read`

```
@ read A
```

valorise la variable A avec le contenu de l'entrée standard

A la fin de ce chapitre, nous verrons plus en détail comment utiliser cette commande **read**.

4.3.2 La commande if

```
@ if commande1
then commande2(s)
else commande3(s)
fi
@
```

⇒ La commande `commande1` est exécutée et par défaut son résultat est affiché à l'écran : si son code de retour est nul alors la suite `commandes2` est exécutée et `commandes3` ne l'est pas.
La partie `else` est optionnelle. Les commandes doivent être séparées par des retours à la ligne ou des ;

⇒ On peut redirectionner la sortie standard de la commande `commande1` si on ne veut pas afficher son résultat à l'écran.

Listing 4.22– Redirection de la sortie de la commande `if`

```
@ if commande1 > res
...
```

Le résultat de commande1 est écrit dans le fichier res

⇒ Il existe des abréviations :

– Le "ET" :

```
@ commande1 && commande2
```

est équivalent à

```
if commande1; then commande2; fi
```

La suite de commandes `commande2` est exécutée si le code de retour de `commande1` est nul. Le code retour est celui de la dernière commande exécutée.

– Le `else` est donné par :

```
commande1 || commande2
```

`commande2` est exécutée uniquement si le code de retour de `commande1` est différent de 0. Le code retour est celui de la dernière commande exécutée (`commande1` ou `commande2`).

Listing 4.23– Commande `if` abrégée

```
@ ls -l fich && echo OK
```

liste le détail de `fich` et si la commande s'est bien terminée, affiche `OK`

Listing 4.24– Commande `if` abrégée

```
ls -l fich || echo KO
```

liste le détail de `fich` et si la commande s'est mal terminée, affiche `KO`

4.3.3 La commande test

La commande **test** permet d'effectuer des comparaisons.

```
test expression
```

ou

```
[ expression ]
```

La commande **test** évalue `expression` et retourne le résultat de cette évaluation (si OK, le code de retour est 0).

Attention il faut un espace avant et après les crochets.

Les différentes options de la commande **test** permettent de réaliser des tests sur les fichiers et répertoires, sur les chaînes de caractères et sur les nombres.

Test sur les fichiers et les répertoires

option	paramètres	description
-w	fichier	vrai si fichier existe et est autorisé en écriture
-r	fichier	vrai si fichier existe et est autorisé en lecture
-x	fichier	vrai si fichier existe et exécutable
-d	fichier	vrai si fichier existe et est un répertoire
-f	fichier	vrai si fichier existe et n'est pas un répertoire
-s	fichier	vrai si fichier existe et a une taille non nulle
-t	descripteur	vrai si descripteur est associé à un terminal

Si "`fichier`" est une chaîne vide la fonction rend 0.

Listing 4.25– Commande `test`

```
@ if test -d monfichier
> then echo Repertoire
> fi
```

Si "monfichier" est un répertoire, on affiche Repertoire , ce qui est équivalent à

Listing 4.26– commande `test`

```
@ if [ -d monfichier]
> then echo Repertoire
> fi
```

Test sur les chaînes de caractères

option	paramètres	description
-z	chaine	vrai si chaine est vide
s1 = s2	s1, s2	vrai si s1 et s2 sont identiques
s1 != s2	s1, s2	vrai si s1 et s2 sont différentes

Listing 4.27– Test sur les chaînes de caractères

```
@ val=bonjour
@ if test -z $val
>    then echo chaine vide
>    else echo chaine non vide
>  fi
chaine non vide
```

Test sur les nombres

option	paramètres	description
n1 -eq n2	n1, n2	vrai n1 = n2
n1 -ne n2	n1, n2	vrai n1 != n2
n1 -gt n2	n1, n2	vrai n1 $>$ n2
n1 -lt n2	n1, n2	vrai n1 $<$ n2
n1 -ge n2	n1, n2	vrai n1 $>=$ n2
n1 -le n2	n1, n2	vrai n1 $<=$ n2

Listing 4.28– Test sur les nombres

```
@ a=1
@ b=2
@ if [ $a -eq $b ] ou if test $a -eq $b
>   then echo egal
>   else echo diff
> fi
diff
```

Si les valeurs des variables `a` *et* `b` *sont égales, on affiche* `egal`*, sinon on affiche* `diff`*.*

Combinaison de tests

On peut combiner tous les tests avec les opérateurs suivants placés à l'intérieur des crochets ou comme paramètres de la commande **test** :

!	pour la négation
-a	pour le **et** logique
-o	pour le **ou** logique

Remarques :
⇒ L'opérateur `-a` est plus prioritaire que l'opérateur `-o`
⇒ Chaque test, opérateur ou opérande doit constituer un mot pour le shell donc être séparé par des espaces.
⇒ Les parenthèses doivent être protégées par un `\` ou encadrées par des ' pour que le shell ne les interprète pas.

Listing 4.29– Combinaison de tests

```
@ if test -d  monRep -a -x monRep ou [ -d monRep -a -x monRep ]
then
  echo chemin accessible
  cd monFich
else
  echo chemin inaccessible
fi
```

Si `monRep` *est un répertoire et s'il a les droits d'exécution, on affiche* `chemin accessible` *et on se déplace dans ce répertoire Sinon on affiche* `chemin inaccessible`*.*

4.3.4 La commande Case

La commande **case** permet de traiter les choix multiples.

```
@ case chaine in
constante1) commande;;
constante2) commande;;
....
constanten) commande;;
esac
```

⇒ La comparaison ne se fait que sous forme de chaînes de caractères.
⇒ Les différentes constantes sont des expressions reconnues par les noms génériques. De plus, on peut éventuellement donner plusieurs modèles pour une seule entrée en ajoutant |. On n'exécute que les entrées qui matchent sinon rien.
⇒ Pour faire une entrée par défaut on peut utiliser *.

Listing 4.30– Commande case

```
@ case $JOUR in
  1) echo Lundi;;
  2) echo Mardi;;
  3) echo Mercredi;;
  4) echo Jeudi;;
  5) echo Vendredi;;
  6) echo Samedi;;
  7) echo Dimanche;;
  *) echo Valeur incorrecte;;
esac
```

Selon la valeur de la variable JOUR qui va de 1 à 7, on affichera le jour en toutes lettres à l'écran. La dernière valeur testée (*) permet de gérer "toutes les autres valeurs".

4.3.5 La commande For

Elle permet l'exécution de commandes identiques sur un ensemble d'élèments.

```
@ for variable in [liste]
  do commande(s)
  done
@
```

Listing 4.31– Exemple de commande `for`

```
@ for i in un deux trois
> do
>    echo $i
> done
 un
 deux
 trois
@
```

Cet exemple aurait pu s'écrire sur une seule ligne :

Listing 4.32– Exemple de commande `for` sur une seule ligne

```
for i in un deux trois; do echo $i; done
```

le " ;" remplaçant le <enter>

4.3.6 La commande While

```
@ while condition
 do commande(s)
 done
```

4.3.7 La redirection des entrés-sorties d'une structure de contrôle

Une structure de contrôle est une commande, de ce fait on peut en rediriger les E/S globales.

Listing 4.33– Redirection de la sortie d'une itération

```
@ for i in *
> do
>   wc -l $i
> done > fichSortie
@
```

Compte le nombre de lignes de chacun des fichiers du répertoire courant (*) et écrit le résultat dans le fichier "fichSortie".

Listing 4.34– Redirection de l'entréé d'une itération

```
@ while read val
>do
> echo $val
>done < fichEntree
```

La variable "val" prend successivement la valeur de chaque ligne du fichier fichEntree : cette procédure affiche à l'écran le contenu de "fichEntrée", ligne par ligne.

4.4 Les variables

Les variables sont des identificateurs dont le nom est une suite suite quelconque de lettres et de chiffres qui ne commence pas par un chiffre (le caractère "_" est admis).

4.4.1 Affectation

Pour affecter une variable on utilise le signe "=" attention, il n'y a pas d'espace !

Listing 4.35– Affectation d'une variable

```
@ var=Bonjour
```

Attribue la valeur "Bonjour" à la variable `var`

4.4.2 Contenu

Le contenu d'une variable est accessible à travers **$**nom_variable.

Listing 4.36– Contenu d'une variable

```
@ X=bonjour
@ echo $X
bonjour
```

Attribue la valeur "bonjour" à la variable X et affiche la valeur de X à l'écran.

Listing 4.37– Affectation de plusieurs valeurs dans une variable

```
@ var=1 2 3 4
bash: 2: command not found
@ var="1 2 3 4"
@ for i in $var
>do
>echo $i
>done
1
2
3
4
```

Affectation des valeurs `1 2 3 4` à la variable `var`. Pour afficher ces différentes valeurs, utilisation d'une boucle `for`

4.4.3 Concaténation de variables

Il est possible de concaténer plusieurs variables dans une seule.

Listing 4.38– Concaténation de variables

```
@ var1="bon"
@ var2="jour"
@ var=$var1$var2
@ echo $var
bonjour
```

4.4.4 Affectation interactive

La commande interne **read**, permet d'assigner une valeur à une variable à partir de l'entrée standard.

Il est possible de saisir plusieurs variables à la fois, dans ce cas il faut donner plusieurs noms de variables à la suite de la commande `read` et utiliser des séparateurs entre les valeurs des variables lors de la saisie (l'espace étant le plus utilisé).

Listing 4.39– Affectation interactive de variables

```
@ read var1
bonjour
@ echo $var1
bonjour
@ read a b
bonjour promo
@ echo $b $a
promo bonjour
@ read var1 var2
bonjour promo info
@ echo $var1
bonjour
@ echo $var2
promo info
@ IFS=o$IFS #définit les séparateurs#
@ read var3 var4
blops
@ echo $var3
bl
@ echo $var4
ps
```

Affectation interactive de `bonjour` *à* `var1`

Affectation interactive de `bonjour` *à* `a` *et* `promo` *à* `b`

Affectation interactive dans les variables `var1` *et* `var2`*. Dans la première variable* `var1` *est affecté le texte saisi jusqu'au premier séparateur rencontré :* `bonjour` *et la suite du texte est affecté à* `var2` *même s'il contient un séparateur*

Ajout d'un nouveau séparateur `o`*. Saisie interactive de* `var3` *et* `var4` *avec la prise en compte de ce nouveau séparateur.*

4.4.5 Expressions arithmétiques

permet de réaliser des calculs avec des nombres entiers.

```
$((nombre1 operation nombre2))
```

où `opération` peut être remplacée par :
`+ - * /` : addition, soustraction, multiplication, division
`%` : reste de la division entière
`**` exponentiel

Listing 4.40– Expressions arithmétiques

```
@ i=1
@ i=$((i+1))
@ echo $i
2
```

Remarque : on n'est pas obligé de préfixer le nom de la variable par $ à l'intérieur des doubles parenthèses.

4.4.6 Portée des variables

La définition d'une variable est limitée au processus (shell par exemple) dans lequel elle a été définie. Elle n'est donc pas connue des processus fils et pères. Il est possible d'étendre la portée d'une variable aux processus fils en l'exportant. La commande **export** transmet une ou plusieures variables aux processus fils du processus courant. Nous reviendrons sur cette notion dans le chapitre concernant les processus.

Listing 4.41– Portée des variables

```
@ Y=toto
@ bash    # creation d'un shell fils
@ echo $Y # sommes dans le fils
@ exit    # sortie du shell fils
exit
@         # retour au pere
```

La variable Y est utilisable dans le shell courant et non connue des processus fils : avec la commande bash un processus shell est créé.

Listing 4.42– Portée des variables

```
$ Y=toto
$ export Y
$ bash    # creation de shell fils
$ echo $Y # dans le fils
$ toto
```

La variable Y est connue dans les processus fils car elle est exportée.

4.4.7 Les variables d'environnement

Outre les variables que l'on décrit soi-même, il existe des variables appelées `variables d'environnement` définies par le système. Toutes ces variables sont utilisables dans les fichiers de commandes et en ligne de commande.

Ces variables prédéfinies n'ont pas besoin d'être exportées, elles sont transmises automatiquement aux processus fils (nous pourrons expliquer ceci à l'aide des chapitres sur les processus et sur la programmation système).

Les commandes **env** et **printenv** listent les variables d'environnement et celles exportées avec la commande `export`.

Le tableau 4.1 donne une liste de quelques variables d'environnement.

Variables	**Descriptif**	**Initialisation - Utilisation**
HOME	Répertoire d'accueil (login)	positionné lors de la création de l'utilisateur
PATH	liste des chemins d'accès aux commandes	idem $HOME ; lorsqu'une commande est tapée, c'est dans ces répertoires qu'elle est recherchée en commençant par le premier : si elle n'est dans aucun, la commande est "inconnue", il faut préciser son chemin pour la lancer
LOGNAME	Nom de connexion (login)	le nom d'utilisateur avec lequel on a ouvert sa session
USER	Nom de l'utilisateur actuel	peut-être différent de LOGNAME si l'utilisateur a été modifié en cours de session
PWD	Répertoire courant	le répertoire dans lequel on se trouve actuellement
SHELL	Nom du shell utilisé	idem $HOME

TABLE 4.1 – Variables d'environnement

4.4.8 Les variables spéciales du shell

Les variables spéciales du shell sont des variables toujours présentes avec un shell, elles sont traitées par le shell. Elles sont accessibles depuis les fichiers de commandes et en ligne de commande.

Le tableau 4.2 donne une liste de quelques variables spéciales du shell.

Variables	Descriptif	Initialisation - Utilisation
IFS	Séparateur de Champs Interne	utilisé pour séparer les mots après les développements et pour découper les lignes en mots avec la commande interne read. La valeur par défaut est `<espace><tabulation><retour-chariot>`
PS1	Prompt primaire	le prompt que l'on a à la connexion ($ en général) auquel on peut ajouter des informations comme le nom de login, le répertoire d'accueil ou courant etc..
PS2	Prompt secondaire	le prompt que l'on a lorsque l'utilisateur a tapé une commande incomplète ou structure de contrôle
RANDOM	Un nombre aléatoire	géré par bash shell et ses dévirés
SECONDS	Temps de connexion en secondes	idem $RANDOM
PID	Identificateur du processus	idem $RANDOM ; numéro du processus du login
PPID	Processus père	idem $RANDOM

TABLE 4.2 – Variables spéciales du shell

La commande set Cette commande, selon les options et arguments utilisés, a différentes conséquences :

1. La commande **set**, donnée sans argument, liste les variables connues du shel avec leurs valeurs.
2. La commande **set** positionne des options pour le shell, entre autres :

   -a : toutes les variables sont exportées

   -e : si erreur sur une commande, arrêt de la procédure

   -u : si les variables n'ont pas de valeur cela génère une erreur

   -v : impression de la ligne sans interprétation des variables

   -x : impression de la ligne avec interprétation des variables

   Pour désactiver ces options : **set +**[option]

Listing 4.43– Positionnement d'options pour le shell avec la commande `set`

```
@ set +u
```

Les variables non valorisées ne provoqueront plus d'erreur

3. La commande **set** permet de positionner la valeur des paramètres positionnels. Cette fonctionalité est présentée dans le paragraphe suivant.

La commande **unset** `nom_variable` permet de supprimer une variable, attention elle peut supprimer une variable d'environnement.

4.4.9 Les paramètres des procédures shell ou paramètres positionnels :

Les fichiers de commandes peuvent être appelés comme les autres commandes, il peut être intéressant de passer des paramètres.

Le fichier de commandes trouve les paramètres d'appel dans des pseudo-variables, appelées **paramètres positionnels** donnés dans le tableau 4.3.

$0	le nom de la procédure
$1	le premier paramètre
$2	le deuxième paramètre
$3	le troisième paramètre
...	
$9	le 9ème paramètre

TABLE 4.3 – Paramètres positionnels

Dans le fichier de commandes, il est possible de consulter le contenu de ces paramètres positionnels mais pas de les modifier directement.

Listing 4.44– Exemple d'appel de procédure shell avec paramètres

```
@ com1 lundi septembre 18
```

Lors de l'appel à cette procédure, les paramètres ont les valeurs suivantes :

```
$0 : com1
$1 : lundi
$2 : septembre
$3 : 18
```

La substitution des méta-caractères a lieu avant l'affectation des paramètres positionnels.

Listing 4.45– paramètres positionnels et substitution des méta caractères

```
@ cat > com1
echo $0 $1 $2
^D
@ ls
fich1  fich2
@ com1 f*
com1 fich1 fich2
```

La commande set permet de positionner les paramètres positionnels.

Listing 4.46– Positionnement des paramètres positionnels avec la commande set

```
@ b="bonjour la promo"
@ set $b
@ echo $1 $2
bonjour la
```

Par défaut, dans la commande `for`, la liste est constituée par la liste des paramètres positionnels.

Listing 4.47– Positionnement des paramètres positionnels avec la commande set

```
@ set 1 2 3
@ for i
> do
> echo $i
> done
1
2
3
```

Listing 4.48– Positionnement des paramètres positionnels avec la commande set

```
@ while read ligne
> do
>  set $ligne
>  echo "deuxieme champs" $2
> done <  fichierTexte
```

Le fichier `fichierTexte` est lu ligne par ligne. La commande `set $ligne` positionne les paramètres positionnels avec les valeurs des différents champs de `$ligne`.

La commande shift permet de déplacer les paramètres vers la gauche. Seul le paramètre `$0` n'est pas affecté. Quand on enchaine plusieurs commandes `shift`, on

recommence toujours à partir de la situation courante et non à partir de celle de départ.

Listing 4.49– Positionnement des paramètres positionnels avec la commande `shift`

```
@ cat > com2
echo $0 $1 $2
shift
echo $0 $1 $2
shift 2
echo $0 $1 $2
^D
@ com2 a b c d e
com2 a b
com2 b c
com2 d e
```

4.4.10 Les informations des procédures shell ou variables automatiques

Ces informations (tableau 4.4) sont initalisées explicitement à chaque procédure.

$@	la liste de tous les paramètres
$#	le nombre de paramètres donnés
$$	le numéro du processus courant
$?	le code retour de la dernière commande passée

TABLE 4.4 – Variables automatiques

Listing 4.50– Exemple de variables automatiques

```
@ ma_procedure lundi septembre 18
```

Lors de l'appel à cette procédure, les variables automatiques ont les valeurs suivantes :

```
$* : lundi septembre 18
$# :  3
```

Le code retour d'une procédure shell est égal à zéro (= 0) si tout s'est bien passé, différent de zéro (!= 0) s'il y a un problème. Ce code retour correspond au code retourné par la commande **exit**.

Lors de la sortie de la procédure, le code retour est celui de la procédure, pas celui de la dernière commande de la procédure !

Dans les exemples suivants nous supposons que le répertoire `/toto` n'existe pas.

Listing 4.51– Exemple de code retour : la procédure maProc1

```
1 ls /toto
2 ERR=$?
3 if [ $ERR -ne 0 ]
4 then exit $ERR
5 else echo ok
6 fi
```

une fois revenu au shell, $? contient le code erreur de la commande "ls /toto"

Listing 4.52– Exemple de code retour : la procédure maProc2

```
1 ls /toto
2 ERR=$?
3 if [ $ERR -ne 0 ]
4 then exit
5 else echo ok
6 fi
```

Une fois revenu au shell, $? contient 0 (le résultat de la commande `maProc2`)

Listing 4.53– Exemple de code retour : la procédure maProc3

```
1 ls /toto
2 ERR=$?
3 if [ $ERR -ne 0 ]
4 then exit 3
5 else echo ok
6 fi
```

Une fois revenu au shell, $? contient 3 (le résultat de la commande maProc3

4.4.11 Substitution de commandes

Il est possible de faire exécuter une commande dans une chaîne ou dans la définition d'une variable, pour cela il faut encadrer la commande par **$(**commande**)** ou `commande` (deux accents graves). Nous utiliserons par la suite la première syntaxe qui est moins source d'erreur de frappe.

Remarques :

⇒ la commande ne doit pas faire de lecture au clavier, elle doit pouvoir être interpretée,

⇒ elle s'exécute dans un processus shell fils : elle peut donc être composée de plusieurs commandes séparées par des ;

⇒ la substitution de commandes permet l'exécution de commandes internes, externes, ou de fichiers de commandes.

Listing 4.54– Substitution de commandes

```
@ logname
dupond
@ a=logname
@ echo $a
logname
@ b=$(logname)
@ echo $b
dupond
@ $(logname)
bash: dupond: command not found
```

La commande `logname` donne le nom de login. La variable a contient la chaîne "logname". La variable b contient le résultat de la commande `logname`

Listing 4.55– Substitution de commandes

```
@ hostname
smith
@ var1=$(logname;hostname)
@ echo $var1
dupond smith
@ pwd
/home/users/dupond
@ var2=$(cd /etc; echo bonjour)
@ echo $var2
bonjour
@ pwd
/home/users/dupond
```

La commande "hostname" donne le nom de la machine. var1 contient le résultat des commandes "hostname" et "logname".var2 contient le résultat des commandes "cd /etc" et "echo bonjour", la commande "cd /etc" s'exécute dans un shell fils, il n'y a donc pas de changement de répertoire dans le shell courant.

Il est possible d'initialiser la liste d'une commande `for` à partir du résultat d'une commande. Dans ce cas, cette commande n'est exécutée qu'une seule fois.

Listing 4.56– Initialisation de la commande `for` avec le résultat d'une commande

```
@ ls
dixparam      fich1          fich2          sortie          status
@ for fich in $(ls)
> do
> echo $fich present
> done
dixparam present
fich1 present
fich2 present
sortie present
status present
```

4.4.12 Les délimiteurs de chaîne

Il existe trois délimiteurs de chaîne de caractères dont l'interprétation est différente :

' (délimiteur habituellement situé sous le 4 , code hexadecimal : 0x27)
`'ceci est une $variable'` : la chaîne de caractères est prise sans substitution

" (délimiteur habituellement situé sous le 3, code hexadecimal : 0x22)
`"ceci est une $variable"` : la chaîne de caractères est prise après substitution de variable

‘ (délimiteur habituellement situé sous le 7 (ALT-GR 7, code hexadecimal : 060)
`ceci est une une ‘$variable‘` : dans ce cas, il faut que $variable contienne une commande : la chaîne de caractères est remplacée par la sortie de la commande. Nous ne conseillons pas d'utiliser cette syntaxe.

➽ *Exemple :*

```
variable=girafe
```

Commande	Résultat
echo '$variable'	$variable
echo "$variable"	girafe
echo ‘ $variable ‘	girafe : command not found

➽ *Exemple :*

```
variable=pwd
```

Commande	Résultat
echo '$variable'	$variable
echo "$variable"	pwd
echo ‘$variable‘	/home/user

Noter que echo $variable rend le même résultat que echo "$variable". En l'absence de délimiteur, la substitution s'effectue.

4.4.13 Les tableaux

Il est possible d'utiliser des tableaux en bash. Comme en langage C, l'indice du tableau commence à 0 et doit être un entier.

Définition et initialisation d'un tableau

Pour créer un tableau il suffit d'initialiser un de ces élèments.

```
nomTableau[indice]=valeur
```

Listing 4.57– Initialisation des élèments d'un tableau

```
@ monTab[0]='Bonjour'
@ monTab[1]='Monsieur'
@ monTab[2]='Madame'
```

On peut aussi initialiser les élèments d'un tableau de cette façon :

```
nomTableau=(valeur1 valeur2 ...)
```

Listing 4.58– Initialisation des élèments d'un tableau

```
@ monTab=(Bonjour Monsieur Madame)
```

Valeur d'un élément d'un tableau

```
${nomTableau[indice]}
```

Listing 4.59– Valeurs des élèments d'un tableau

```
@ echo ${monTab[0]}
Bonjour
```

Caractéristiques d'un tableau

⇒ Nombre d'élèments d'un tableau (seuls les élèments initialisés sont comptabilisés) :

```
${#nomTableau[@]}
```

Listing 4.60– Nombre d'élèments d'un tableau

```
@ echo ${#monTab[@]}
3
```

⇒ La liste de tous les éléments d'un tableau :

```
${nomTableau[@]}
```

Listing 4.61– Liste de tous les éléments d'un tableau

```
@ echo ${monTab[@]}
Bonjour Monsieur Madame
```

⇒ La liste de tous les indices d'un tableau :

```
${!nomTableau[@]}
```

Listing 4.62– Liste de tous les indices d'un tableau

```
@ echo ${!monTab[@]}
0 1 2
```

Suppression d'un tableau

```
unset nomTableau
```

Suppression d'un élément d'un tableau

```
unset nomTableau[indice]
```

Parcourir un tableau

Listing 4.63– Parcours d'un tableau à partir de ses indices

```
@ for i in "${!monTab[@]}"; do echo ${monTab[$i]}; done
Bonjour
Monsieur
Madame
```

Listing 4.64– Parcours d'un tableau à partir de ses éléments

```
@ for element in "${monTab[@]}"; do echo $element; done
Bonjour
Monsieur
Madame
```

Listing 4.65– initialisation d'un tableau à partir d'un fichier

```
@ cat fich1
un
deux
trois
quatre
@ i=0; while read line ; do monTab[$i]=$line; i=$((i+1)); done < fich1
@ echo ${monTab[1]}
deux
```

4.5 Les filtres, les pipes et la manipulation de fichiers

4.5.1 Les filtres

Les filtres sont des commandes qui traitent des données de l'entrée standard ou d'un fichier pour délivrer un résultat.

Le résultat des filtres peut être redirigé dans un fichier.

Ces commandes (et toutes celles de ce cours !) ont pour la plupart des options qui ne sont pas détaillées ici mais que vous pouvez visualiser à l'aide de la commande **man**.

La commande tr

Elle permet trois types de transformation sur l'entrée standard :

⇒ transcodage : commande `tr` sans option. Elle permet de remplacer une chaîne de caractères par une autre. Elle lit et respectivement, écrit sur l'entrée et la sortie standard.

```
tr chaine1 chaine2
```

⇒ suppression de certains caractères : commande `tr` avec l'option `-d`.

```
tr -d chaine1
```

⇒ suppression de répétitions : commande `tr` avec l'option `-s`.

```
tr -s chaine1
```

Listing 4.66– Exemple commande tr

```
@ tr 'a' 'A' <fich1 >fich2
```

Remplace les caractères `'a'` du fichier `fich1` par des caractères `'A'`. Le résultat est redirigé sur le fichier `fich2`.

Listing 4.67– Exemple commande tr

```
@ tr 'a-z' 'A-Z' <fich1 >fich2
```

Remplace les caractères miniscules du fichier `fich1` par des caractères majuscules. Le résultat est redirigé sur le fichier `fich2`.

Listing 4.68– Exemple commande tr

```
@ tr -d '' <fich1 >fich2
```

Supprime les espaces dans le fichier `fich1`. Le résultat est redirigé sur le fichier `fich2`.

Listing 4.69– Exemple commande tr

```
@ tr -s '\012' <fich1 >fich2
```

Remplace plusieurs "retour à la ligne" par un seul dans le fichier `fich1`. Le résultat est redirigé sur le fichier `fich2`.

La commande sort

Elle trie les données en entrées : par défaut, par ordre croissant, sur le premier champ du fichier.

Listing 4.70– Tri sur le second champs du fichier

```
@ sort -k2 nomFichier
```

Listing 4.71– Tri en ordre inverse (décroissant) sur le 2nd champ du fichier

```
@ sort -k2r nomFichier
```

Listing 4.72– Exemple commande `sort`

```
@ cat MonFichierdeCDE
ma première ligne
ma deuxième ligne
ma troisième ligne
et la dernière
@ sort MonFichierdeCDE
et la dernière
la deuxième ligne
ma première ligne
ma troisième ligne
```

La commande colrm

Elle supprime des colonnes sélectionnées de l'entrée standard.

```
colrm [debut [fin]]
```

Si seulement `debut` est spécifié : seules les colonnes inférieures en numéro à `début` sont affichées.
Si `debut` et `fin` sont spécifiés : les colonnes inférieures en numéro à `début` et les colonnes supérieures en numéro à `fin` sont affichées.

Listing 4.73– Exemple commande `colrm`

```
@ ls-l
drwxr-xr-x  2 root root  4096 oct  9  2009 bin
drwxr-xr-x  3 root root  4096 oct  9  2009 boot
lrwxrwxrwx  1 root root    11 oct  9  2009 cdrom -> media/cdrom
drwxr-xr-x 12 root root  3200 nov 13 15:18 dev
drwxr-xr-x 44 root root  4096 nov 13 15:18 etc
drwxr-xr-x  4 root root  4096 oct  9  2009 home
@ ls -l | colrm 25
drwxr-xr-x  2 root root
drwxr-xr-x  3 root root
lrwxrwxrwx  1 root root
drwxr-xr-x 12 root root
drwxr-xr-x 44 root root
drwxr-xr-x  4 root root
```

La commande cut :

Elle extrait des colonnes ou des champs d'un fichier :

cut -c *liste* : extrait les colonnes dont les numéros sont donnés dans liste (séparés par des virgules, ou des intervalles avec -).
cut -f *liste* **-d** *délimitateur* : extrait les champs dont les numéros sont donnés dans liste. Ces champs sont délimités par " délimitateur". Si celui ci n'est pas spécifié, on utilise celui stocké dans la variable d'environnement `IFS`.

Listing 4.74– Exemple commande `cut`

```
@ ls -l | cut -c 17-25
grossin
grossin
grossin
grossin
grossin
grossin
grossin
```

Extrait les colonnes 17 à 25 du résultat de la commande ls -l

Listing 4.75– Exemple commande `cut`

```
@ cat exCut
message 1 : 1er jour : lundi
message 2 : 2eme jour : mardi
message 3 : 3eme jour : mercredi
@ cut -f1,3 -d: exCut
message 1 : lundi
message 2 : mardi
message 3 : mercredi
```

Extrait les champs n°1 et 3 délimités par " :" du fichier "exCut"

La commande uniq :

Elle supprime les lignes consécutives identiques dans un fichier.

Listing 4.76– Exemple commande `uniq`

```
@ cat  fic1
1
1
1
2
3
3
4
5
@ uniq fic1
1
2
3
4
5
```

4.5.2 Les expressions régulières et la commande grep

Unix offre des outils (ex : grep, sed, awk) permettant de manipuler du `texte` contenu dans des fichiers. Ces outils utilisent des expressions régulières. Les expressions régulières sont formées à l'aide de caractères spéciaux.

Les caractères spéciaux dans les expressions régulières

.	un caractère quel qu'il soit, s'il n'est pas entre crochets
[ ]	un des caractères inscrits entre les crochets
[^]	tout sauf un des caractères entre les crochets
^	expression recherchée est en début de ligne
$	expression recherchée est en fin de ligne

TABLE 4.5 – Expr. régulières : caractères spéciaux

Remarque :
Les caractères spéciaux n'ont pas toujours la même signification dans le développement des noms de fichiers et dans les expressions régulières (permettant, elles, la manipulation de texte à l'intérieur d'un fichier). Vous trouverez en Annexe un tableau récapitulatif de ces caractères avec leurs différentes utilisations.

➽ *Exemple :*

```
[a b c] : a b ou c
[a-g]   : un caractère de a à g
[^ q ]  : tout caractère autre que q
```

```
[. p]    : le point ou p
.p       : n'importe quel caractère devant un p
```

On peut utiliser des opérateurs de répétition dans les expressions régulières (tableau 4.6).

*	0 ou plusieurs fois
?	0 ou une fois
+	une ou plusieurs fois
{n}	exactement n fois
{n,}	n fois ou plus
{,m}	au maximum m fois

TABLE 4.6 – Expr. régulières : op de répétition

Remarques :
⇒ Le caractère \ permet de supprimer le sens spécial du caractère qui le suit.
⇒ La concaténation de deux expressions régulières se fait par juxtaposition.
⇒ L'alternative entre expressions régulières est donnée par l'opérateur |.
⇒ Le groupage d'expressions régulières se réalise à l'aide de parenthèses.

➽ *Exemple :*

```
[A-Z]?      une majuscule 0 ou une fois
a*          zéro ou n fois la lettre a
abc         la chaîne abc
[Oo]ui      la chaîne Oui ou oui
[A-Z][0-9]          une  majuscule et un chiffre
bonjour|bonsoir     la chaîne bonjour ou bonsoir
[A-Z]{8}|bonjour    correspond à 8 lettres ou la chaîne bonjour
oui                 correspond à la chaîne oui
oui{2}              correspond à la chaîne ouii
(oui){2}            correspond à la chaîne ouioui
```

La commande grep

Cette commande recherche une expression, exprimée à l'aide d'expressions régulières, dans un fichier. Elle affiche les **lignes** du fichier contenant l'expression recherchée.

Listing 4.77– Exemple de commande `grep`

```
@ cat MonFichierdeCDE
ma première ligne
ma deuxième ligne
ma troisième ligne
et la dernière
@ grep ière MonFichierdeCDE
ma première ligne
et la dernière
```

affiche les lignes où se trouve l'expression "ière"

Lors de la recherche d'expressions régulières un peu complexes, on utilise l'option `-E` de la commande `grep` et les expressions régulières doivent être mises entre quote (" ").

```
grep -E "[A-Z]{8}| bonjour"
```

Exemples :

1. afficher toutes les lignes commençant par la lettre "t" du fichier `monFich` du répertoire :

```
grep ^t monFich
```

2. afficher toutes les lignes finissant par la lettre "c" du fichier `monFich` du répertoire :

```
grep c$ monFich
```

3. afficher toutes les lignes contenant 0 ou plusieurs a du fichier `monFich` du répertoire :

```
grep a* monFich
```

4. afficher le contenu de tous les fichiers dont le nom ne commence pas par f

```
cat [!f]*
```

5. afficher les lignes se terminant par * des fichiers dont le nom commence par test

```
cat test* | grep \*$
```

4.5.3 Les pipes ou tubes

Ils consistent à diriger la sortie standard d'une commande sur l'entrée standard d'une autre commande.

Le symbole permettant la création d'un pipe entre deux commandes est le | (Alt Gr 6 sur la plupart des claviers).

En reprenant les notions d'entrée standard (descripteur de fichiers égal à 0) et de sortie standard (descripteur de fichier égal à 1) d'une commande (ou processus), on arrive au schéma suivant :

```
cde1 --> desc 1 |__________| desc 0 --> cde2
```

La sortie de la commande cde1 (son descripteur 1) n'est pas dirigée sur la sortie standard (écran) mais alimente l'entrée (le descripteur 0) de la commande cde2.

Reprenons le fichier "MonFichierdeCDE". On affiche les lignes de ce fichier contenant l'expression "ère" triées par ordre croissant.

```
@ grep ière MonFichierdeCDE > ./temp
@ sort ./temp
@ rm ./temp
```

Un fichier intermédiaire ./temp est créé, qui restera sur le disque s'il n'est pas supprimé (d'où l'ajout de la ligne rm ./temp).

```
@ grep ière MonFichierdeCDE | sort
```

Dans ce cas, aucun fichier intermédiaire n'est créé.

Si le résultat dépasse le nombre de lignes de mon écran, je peux écrire :

```
@ grep ière Mon_Fichier_de_CDE | sort | more
```

Dans ce cas, la pagination se fait écran par écran.

Ecrire un shell dans lequel on supprime tous les fichiers ordinaires du répertoire courant sans toucher aux répertoires, pour ce faire on écrit le shell suivant :

```
1 ls | while read FICH
2 do
3    if [ -f $FICH ]
4    then
5     rm $FICH
6     echo suppression du fichier $FICH
7    fi
8 done
```

La première ligne exécute la commande ls qui a comme sortie une suite de noms de fichiers (ordinaires ou non). Cette sortie est redirigée en entrée de la boucle while .

Dans la boucle while, on lit (read) chacune des informations (noms de fichiers) et on met le résultat de cette lecture dans une variable FICH.

Pour chaque ligne (nom de fichiers) , on teste si le contenu de la variable FICH est un fichier ordinaire(-f $FICH), si c'est le cas on supprime le fichier dont le nom est contenu dans FICH et on affiche un message en sortie. Comme rien n'est précisé, ce message sera affiché sur la sortie standard (écran).

4.5.4 Manipulations de fichiers

Nous venons de voir différentes commandes qui permettent de visualiser des noms de fichiers ou des contenus de fichiers en appliquant des règles. Nous voyons ici d'autres commandes qui vont permettre de gérer les fichiers. Nous faisons un rappel de quelques commandes vues au chapitre 2, avant d'en introduire de nouvelles.

La commande more

Elle permet de visualiser le contenu d'un fichier en paginant à chaque remplissage d'écran (le filtre est la pagination).

Listing 4.78– Exemple de commande `more`

```
@ more FichierTexte
```

affiche le fichier FichierTexte à l'écran en interrompant l'affichage à chaque remplissage de l'écran

La commande cat

Elle permet d'afficher le contenu d'un ou plusieurs fichiers sur la sortie standard. S'il y a plusieurs fichiers, ils sont assemblés en une seule sortie à l'écran (le filtre est la concaténation) - Cette commande est surtout utilisée avec la redirection de sortie standard afin de créer un fichier contenant la totalité des fichiers passés en argument.

Listing 4.79– Exemple de commande `cat`

```
@ cat Fichier1 Fichier2 Fichier3 > FichierTotal
```

FicherTotal est constitué du contenu des 3 fichiers à la queue leu leu

La commande wc

La présentation de cette commande et des exemples sont donnés au chapitre 2.

La commande paste :

Elle réalise la fusion de n fichiers par concaténation des lignes l'une à côté de l'autre, le résultat est transmis à la sortie standard. Elle permet aussi d'afficher un fichier en multi-colonnes.

Listing 4.80– Exemple de commande `paste`

```
@ cat fic1
1
2
3
@ cat fic2
1
2
3
4
5
@ paste fic1 fic2 > ficNew
@ cat ficNew
1 1
2 2
3 3
  4
  5
```

La commande od :

Elle affiche le contenu d'un fichier dans différents formats à partir des codes ASCII. Différentes option de la commande `od` :

⇒ **-b** affiche le contenu du fichier en octal

⇒ **-x** affiche le contenu du fichier en hexadécimal

⇒ **-c** affiche le contenu du fichier sous la forme de caractères

Tout le contenu du fichier est affiché, y compris les espaces, les fins de ligne, etc.

Listing 4.81– Exemple de commande `od`

```
@ cat fic1
1
2
3
4
5
@  od -c fic1
0000000    1  \n   2  \n   3  \n   4  \n   5  \n
0000012
@ od -b fic1
0000000   061 012 062 012 063 012 064 012 065 012
0000012
@ od -x fic1
0000000 0a31 0a32 0a33 0a34 0a35
0000012
```

Remarque : le code ASCII de fin de ligne d'un fichier UNIX est Oa (LF ou line feed). Ce n'est pas le même sous DOS (OdOa).

La commande tee :

Elle recopie son entrée sur la sortie standard et dans un fichier : elle s'utilise dans un pipe (figure /reftee).

Listing 4.82– Exemple de commande `tee`

```
@ ls | tee mon_fichier_detail
```

La liste des fichiers du répertoire est affichée à l'écran et copiée dans le fichier mon_fichier_detail. Ce fichier est créé ou écrasé s'il existait déjà. Si l'on veut écrire à la suite dans un fichier existant, il faut utiiser l'option -a

Listing 4.83– Exemple de commande `tee`

```
ls | tee mon_fichier_detail | wc -l
```

dans mon_fichier_detail il y a la liste des fichiers du répertoire (résultat de la commande ls), à l'écran le nombre de fichiers dans ce répertoire (résultat de la commande wc appliqué à ls)

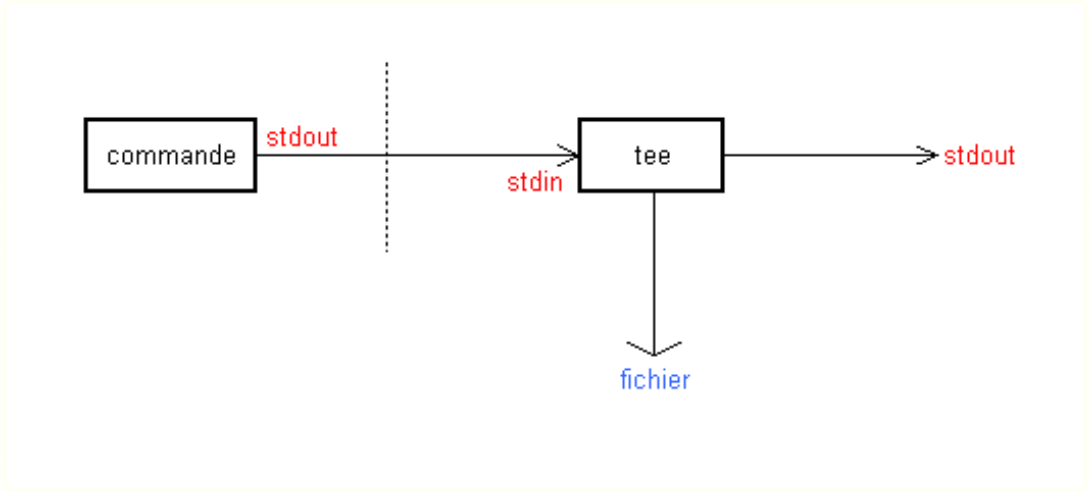

FIGURE 4.1 – Illustration de la commande tee

4.5.5 La commande find

La commande **find** est une commande particulière qui permet la recherche de fichiers dans une arborescence et le filtrage des fichiers rencontrés. Cette commande accepte de nombreuses options dont nous allons lister les plus couramment utilisées.

```
find répertoire [répertoires] expression
```

où :
repertoire : est le répertoire dans lequel s'effectue la recherche.
expression est obtenue en combinant des `options` (tableau 4.7) à l'aide d'`opérateurs` (tableau 4.8). Lorsqu'on fait référence à une valeur numérique, on note **n** pour la valeur **n**, $+$**n** pour une valeur $>$**n**, $-$**n** pour une valeur $<$**n**.

Lorqu'on précise une commande avec l'option `-exec`, le symbole {} symbolise chaque fichier sur lequel cette commande sera exécutée.

Listing 4.84– Exemple de commande `find`

```
@ find . \( -name "test*" -o -name essai \) -ctime 0 -exec rm {} \;
```

Décomposons cette commande :
⇒ *on recherche à partir du répertoire courant (.) et dans ses sous répertoires*
⇒ *on recherche les fichiers dont le nom commence par `test` (notez que le caractère générique * implique l'utilisation de quotes) ou (disjonction* ***-o*** *) s'appelant `essai`*
⇒ *et dont la date de création est aujourd'hui (****-ctime*** *0 : zéro jours depuis la création)*

Options	Signification
-name	nom du fichier
-inum	numéro d'inode
-type	b,c,d,f,l
-links	nombre de liens
-user	nom du propriétaire
-group	nom du groupe
-size	taille en blocs (1024 Caractères sous Linux)
-atime	nombre de jours depuis la dernière consultation
-mtime	nombre de jours depuis la dernière modification
-ctime	nombre de jours depuis la création
-print	affiche le résultat
-exec	commande shell à excécuter pour chaque fichier concerné
-ok	idem exec avec confirmation

TABLE 4.7 – Les options de la commande find

Opérateurs	Combinaisons
juxtaposition	conjonction
-o	disjonction
!	négation
\(. . . . . \)	associativité

TABLE 4.8 – Opérateurs de combinaison (commande find)

⇒ *on exécute la commande **rm** pour tous les fichiers répondant à ces critères .*
Rappel : les \ permettent de ne pas passer au shell des caractères qu'il interpréterait.
En effet les parenthèses sont à passer à la commande find et le point virgule est à passer avec la commande rm pour chaque fichier.

4.6 Exercices

Exercice 2 : Utilisation de commandes

1. Donner une commande qui affiche la liste de tous les fichiers présents dans l'arborescence de l'utilisateur qui l'exécute.
2. Donner une commande qui affiche tous les fichiers présents dans les répertoires `/bin` et `/lib`.
3. Donner une commande qui liste les fichiers du répertoire `texte` de l'utilisateur `dupond` en n'affichant que les noms de répertoires. Puis, donner une commande qui n'affiche que les fichiers ordinaires de ce même répertoire.
4. Donner une commande qui affiche tous les fichiers dont le nom commencent par "fic" du répertoire `texte` de l'utilisateur `dupond`.
5. Donner une commande qui recherche dans toute l'arborescence du système, les fichiers ayant une taille supérieure à 10 Mo ou ayant les droits d'accès 4755. Supprimer les éventuels messages d'erreurs avec une redirection.
6. Donner une commande qui affiche le format de chaque fichier dont le nom commence par la lettre "p" dans l'arborescence du répertoire `/etc`.
7. Donner une commande qui affiche les lignes du fichier `dupond/texte/fich1.tex` contenant la chaîne "http".

Exercice 3 : Fichier de commandes : exécution - code retour - paramètres

1. Ecrire un shell script `shell1` qui effectue les opérations suivantes :
   ⇒ affecter la valeur "bonjour" à la variable `var`.
   ⇒ afficher "la variable var a pour valeur : " suivi de la valeur de `var`.
   ⇒ effectuer une pause de 3 secondes.
2. Exécuter le shell script de la manière suivante :
   `bash shell1`
   A la fin de l'exécution du shell script, quelle est la valeur de la variable `var` dans votre shell ? pourquoi ?

3. Exécuter le shell script de la manière suivante :
   `shell1`
   Est ce que c'est possible ? Que faut-il faire pour pouvoir exécuter le shell script de cette façon ? Réaliser les modifications et exécuter le shell script. A la fin de l'exécution du shell script, quelle est la valeur de la variable `var` dans votre shell ? pourquoi ?
4. Exécuter le shell script de la manière suivante :
   `. shell1`
   A la fin de l'exécution du shell script, quelle est la valeur de la variable `var` dans votre shell ? pourquoi ?
5. Avant d'exécuter le shell script, initialiser une variable dans votre shell : `varPere="pere"`. Ajouter la ligne suivante dans le shell script `shell1` :
   `echo "valeur de la variable varPere: $varPere"`
   Exécuter le shell selon les trois manières proposées précédemment. Quelles sont conclusions ? Maintenant, exporter la variable `varPere` avant de lancer les exécutions de `shell1`.Quelles sont vos conclusions ?
6. Recopier le shell script `shell1` dans `shell2`. Modifier shell2 de la façon suivante :
   ⇒ lire deux variables et les afficher.
   ⇒ afficher un code de retour égal à 2.
   Exécuter ce nouveau shell script et afficher la valeur du code de retour.
7. Ecrire un shell script `shell3` qui :
   ⇒ affiche le nombre de paramètre du shell script et son nom.
   ⇒ teste si le nombre de paramètres positionnels est égal à 2, et affiche le résultat du test.
   ⇒ affiche les trois premiers paramètres positionnels du shell script.

Exercice 4 : Fichier de commandes : la commande `set`

Question 1 Ecrire un shell script qui affiche le mois courant en utilisant la commande `date` sans argument.

Question 2 Ecrire un shell script qui affiche les noms de fichiers du répertoire courant qui ont été modifiés durant le mois.

Exercice 5 : Fichier de commandes : sauvegarde

Question 5.1 : Créez sous votre répertoire d'accueil un répertoire EXO1 dans lequel vous créerez manuellement 10 fichiers nommés UN à DIX. Chaque fichier contient une

ligne, par exemple : le fichier UN contient "Première ligne", le fichier DEUX contient "Deuxieme ligne", etc.

Question 5.2 : Créez un fichier de commandes exécutable par tous les membres de votre groupe qui :

1. crée sous le répertoire d'accueil de l'utilisateur qui lance le script un sous répertoire de nom "siècle an mois jour " (par ex : 20070102 pour le 2 février 2007)
2. recopie les fichiers de EXO1 sous ce répertoire puis les enlève de EXO1
3. crée sous le répertoire d'accueil de l'utilisateur qui lance le script deux fichiers :
   ⇒ un fichier nommé "gros_fichier.numero_du_shell" dans lequel se trouvera le contenu concaténé des fichiers traités.
   ⇒ un fichier nommé "nom_de_la_procedure.numero_du_shell" dans lequel se trouvera le nom des fichiers traités.

Indications :
⇒ voir la commande **date** pour créer le répertoire
⇒ vérifier que le répertoire à créer n'existe pas déjà

Exercice 6 : Fichier de commandes : date de dernière modification

Créez un fichier de commandes `change` qui affiche la date de dernière modification d'un fichier puis la modifie avec l'heure actuelle. Ce fichier de commandes prend comme paramètre le nom du fichier (chemin absolu).

Exemple :

```
$ change mon_fic
```

Le 8 octobre à 15 heures 12 vous aurez le résultat :

```
-r--r--r-- 1 user group 40 Fev 3 2001 mon_fic
-r--r--r-- 1 user group 40 Oct 8 15:12 mon_fic
```

⇒ voir la commande **ls** pour afficher la date de dernière modification.

Exercice 7 : Fichier de commandes : nombre de jours du mois

Créez une procédure `nombre` qui affiche le nombre de jours du mois courant.

Exemple :

```
$ nombre
```

En février 2009, nous aurons le résultat suivant :
`Il y a 28 jours en février`

⇒ voir la commande **cal** pour afficher un calendrier.

4.7 Correction des exercices

Correction Exercice 2 : Utilisation de commandes

1. `find ~/. -print`
2. `find /bin /lib -print`
3. `find ~dupond/texte -type d`
   `find ~dupond/texte -type f`
4. `find ~dupond/texte -name "fic*"-print`
   `fic*` est mis entre " pour indiquer que le shell ne doit pas l'interpréter, l'étoile doit être interprétée par la commande `find`.
5. `find / -size +10000k -o -perm 4755 -print 2>/dev/null`
6. Correction
   ```
   1 find /etc -name "p*" -print -exec file {} \;  2>/dev/null}
   ```
7. `grep "http"dupond/texte/fich1.tex`
   avec la commande `grep`, on recherche des inforamations dans des fichiers texte.

Correction Exercice 3 : Fichiers de commandes : exécution - code retour - paramètres

1. Correction `shell1`

   Listing 4.85– shell1
   ```
   1 var="bonjour"
   2 echo "la variable var a pour valeur: $var"
   3 sleep 3
   ```

2. La variable `var` n'est pas connue du shell courant car le shell script `shell1` a été exécuté dans un autre shell.

3. Ce n'est pas possible d'exécuter `shell1` comme une commande car le shells-cript n'a pas les droits d'exécution. Il faut lui attribuer avec la commande **chmod**.

```
@ chmod u+x shell1
```

   On peut alors exécuter `shell1`. Il faut au préalable vérifier que la variable PATH contient bien le "." (point) ou donner le chemin d'accès de la commande `shell1`.

   La variable `var` n'est pas connue du shell courant car le shell script `shell1` a été exécuté dans un autre processus que le processus courant.

4. Le shell courant connait la valeur de la variable `var` à l'issue de l'exécution de `shell1` avec la commande **.** car `shell1` a été exécutée dans le shell courant.

5. La variable `varPere` est connue uniquement dans le shell courant, elle n'est donc connue par `shell1` uniquement quand il s'exécute dans le shell courant à l'aide de la commande **.**

   Une variable qui est exportée est connue de tous les processus créés par le processus courant. Ce qui explique que dans notre exemple, après avoir réalisé la commande `export varPere`, cette variable est connue dans le `shell1` quelque soit la façon dont on l'exécute.

6. Correction `shell2`

Listing 4.86– shell2

```
1 var="bonjour"
2 echo "la variable var a pour valeur: $var"
3 echo "la variable varPere a pour valeur: $varPere"
4 sleep 3
5 echo "entrez une valeur"
6 read a
7 echo "entrez une seconde valeur"
8 read b
9 echo "les valeurs saisies sont $a et $b"
10 exit 2
```

7. Correction `shell3`

Listing 4.87– shell3

```
1 echo "nombre de parametre: $#"
2 echo "nom de la commande: $0"
3 if [ $# -ne 2 ]
4 then
5 echo "nb parametres superieur a deux"
6 fi
7 echo "les trois premiers parametres sont: $1 $2 $3"
```

   Exécutez ce shell en lui passant des paramètres.

Correction Exercice 4 : Fichier de commandes : la commande `set`
Solution question 4.1 : Ecrire un shell script qui affiche le mois courant en utilisant la commande `date` sans argument.

```
1 a=$(date)
2 set $a
3 echo "mois courant: " $3
```

Solution question 4.2 : Ecrire un shell script qui affiche les noms de fichiers du répertoire courant qui ont été modifiés durant le mois.

```
1 set $(date)
2 mois=$3
3 ls -l | while read line
4 do
5 set -- $line
6 if [ "$7" =  "$mois" ]
7 then
8 echo $9
9 fi
10 done
```

Indications :

⇒ `set --` : les deux "-" à la suite de la commande `set` indiquent qu'elle ne doit pas considérer la suite comme des options. Si on ne les met pas, lorsque le premier caractère d'une ligne résultat de la commande `ls -l` est un "-", elle prend la suite comme une option. A tester.
⇒ on ne peut pas utiliser la commande `cut` pour récupérer le champs mois dans le résultat de la commande `ls -l` car les champs ne sont pas délimités par un nombre fixe d'espaces. A tester.

Correction Exercice 5 : Fichier de commandes : sauvegarde
Solution question 5.1 :

Création du répertoire

```
mkdir $HOME/EXO1
```

Modification des droits d'accès

```
chmod o-rx $HOME/EXO1
```

Création des dix fichiers

```
cat > $HOME/EXO1/UN
Premiere ligne
CTRL-D

cat > $HOME/EXO1/DEUX
Deuxieme ligne
CTRL-D

.
.
.
cat > $HOME/EXO1/DIX
Dixieme ligne
CTRL-D
```

Solution question 5.2 :

Création du shell (vous aurez sans doute à faire des aller/retour sous vi pour sa mise au point !)

Listing 4.88– Sauvegarde

```
1  cat > SHEL1
2
3  # Positionnement de la variable DATE avec ssaammjj
4  DATE=$(date +%Y%m%d)
5
6  # Le répertoire n'existe pas on le crée
7  if [ ! -d $HOME/$DATE ]
8     then mkdir $HOME/$DATE
9  fi
10 #changement des autorisations sur le répertoire
11 chmod o-rx,g+w $HOME/$DATE
12
13 # boucle sur la commande ls
14 ls $HOME/EXO1 | while read fichier
15 do
16    cp $fichier $HOME/$DATE
17    if [ $? -ne 0 ]
18       then exit
19       else rm $fichier
20     fi
21    cat $HOME/$DATE/$fichier >> $HOME/gros_fichier.$$
22    basename $HOME/$DATE/$fichier >> $HOME/$0.$$
23
24 done
```

Autorisation d'exécuter pour le groupe et modification juste par moi :

```
chmod g=rx,u=rwx nomRep
```

Correction Exercice 6 : Fichier de commandes : date de dernière modification

L'affichage de la date de dernière modification d'un fichier se fait par la commande **ls -lc**

Listing 4.89– Date de dernière modification

```
1 echo $(ls -lc $1)
2 ln $1 tmp
3 rm tmp
4 echo $(ls -lc $1)
```

Remarque : dans cette correction on utilise la commande `ln` qui est présente sur la majorité des sytsèmes UNIX. On peut aussi utiliser la commande `touch`.

Correction Exercice 7 : Fichier de commandes : nombre de jours du mois

La commande **cal** affiche le calendrier du mois et de l'année passés en parametres.

Listing 4.90– Nombre de jours du mois

```
1 AN=$(date +%Y)
2 MS=$(date +%m)
3 RES=$(cal $MS $AN |wc -w)
4 RES1=$(($RES - 9))
5 case $MS in
6       1) MOIS=Janvier;;
7       2) MOIS=Fevrier;;
8       3) MOIS=Mars;;
9       4) MOIS=Avril;;
10      5) MOIS=Mai;;
11      6) MOIS=Juin;;
12      7) MOIS=Juillet;;
13      8) MOIS=Aout;;
14      9) MOIS=Septembre;;
15     10) MOIS=Octobre;;
16     11) MOIS=Novembre;;
17     12) MOIS=Decembre;;
18 esac
19 echo Il y a $RES1 jours en $MOIS
```

Chapitre 5

Gestion des utilisateurs -GL-

Dans ce quatrième chapitre, vous verez comment gérer plusieurs utilisateurs.

5.1 La gestion multi utilisateurs

Unix est un système **multi utilisateurs** : Il est possible à plusieurs utilisateurs de se connecter et d'utiliser la même machine simultanément, aussi bien de manière local que distante.

Afin de permettre à chacun de travailler avec les autres mais sans interférence incontrôlée, les utilisateurs sont décrits dans des fichiers que nous allons explorer dans cette leçon. Nous avons déjà ébauché des concepts dans le chapitre 2 sur les fichiers, ces notions sont donc ici présentées plus en détail quant à leur mise en oeuvre.

5.1.1 L'utilisateur

L'utilisateur est connu par son nom de connexion (la réponse à **login :**). Ce nom est unique et est associé à un numéro, le **UID** (**U**ser **ID**entify).

Ce numéro sera utilisé par le système dans toutes les tables qu'il gère.

> *un utilisateur particulier permet de gérer le système avec TOUS LES DROITS sur TOUS LES FICHIERS :* ***root***

Cet utilisateur est encore appelé "super utilisateur" et existe sur tous les systèmes. Il est déconseillé de se logguer avec cet identifiant, même si on utilise une machine déconnectée d'un réseau. En effet, ayant tous les droits, cet utilisateur a entre autres la possibilité de détruire TOUS les fichiers de votre système, y compris ceux du système lui-même (ce qui est dangereux)

Les utilisateurs sont décrits dans un fichier : **/etc/passwd** dans lequel chaque enregistrement est découpé de la façon suivante

nom celui qui sera utilisé comme login

mot de passe apparaît sous forme cryptée

numéro UID identifiant unique du nom

numéro GID numéro du groupe principal auquel appartient le nom

commentaire généralement, le nom en clair de la personne

répertoire répertoire d'accueil (deviendra $HOME)

programme à lancer à l'ouverture de session le chemin du programme à exécuter au moment de l'ouverture d'une session ; le shell pour un informaticien, tout autre programme pour un utilisateur

5.1.2 Le groupe

A l'instar de l'utilisateur , le groupe est connu par un nom et par un numéro unique dans le système le **GID** (**G**roup **ID**entity)

L'utilisateur appartient à un groupe *principal*, et peut appartenir à des groupes *secondaires* . Les groupes sont décrits dans un fichier **/etc/group** avec le découpage suivant :

nom du groupe nom qui pourra être utilisé pour les attributions de droits nom du groupe

mot de passe mot de passe d'accès au groupe : apparaît sous forme cryptée (très peu utilisé)

numéro GID identifiant unique du groupe

utilisateurs du groupe liste des "noms" du fichier /etc/passwd appartenant au groupe

Le UID "0" est attribué au super utilisateur - Le GID "0" signifie "groupe administrateur"

Cette signification du "zéro" est figée. Vous pourriez par exemple décider que sur votre machine le user root n'existe pas, et le nommer superman : l'attribution de l'IUD "0" à "superman" le rend super utilisateur (dans la pratique personne ne fait cela ... !) Exemple : Nous avons 2 utilisateurs, Alain et Paul et 3 groupes : études (etu), projet (pro) et exploitation (exp).

Listing 5.1– Exemple de contenu de fichier `/etc/passwd`

```
alain::110:500:Alain Terrieur:/home/alain:/bin/ksh
paul::120:500:Paul Hissont:/home/paul:/bin/ksh
```

Listing 5.2– Exemple de contenu de fichier `/etc/group`

```
etu::500:alain,paul
pro::510:alain,paul
exp::520:alain
```

Le groupe principal des 2 utilisateurs est "etu". Lors de la création d'un fichier par ces utilisateurs, ils seront créés par défaut comme appartenant à cet utilisateur et à son groupe principal.

Alain Terrieur crée un fichier fic1. Il positionne les permissions suivantes :

```
-rw-rw---- 1 fic1 alain etu .....................
```

ce fichier est créé comme appartenant à **alain**, et dépendant du groupe **etu** (groupe principal d'alain) Alain Terrieur crée deux autres fichiers, qu'il modifie quant à leur groupe, avec les permissions suivantes

```
-rw-r----- 1 fic2 alain pro .....................
-rw-rw---- 1 fic3 alain exp .....................
```

Paul Hissont pourra :
⇒ lire et modifier fic1 (Paul appartient au groupe etu et fic1 est en lecture/écriture pour les membres du groupe), il pourra
⇒ lire fic2 (Paul appartient au groupe pro et fic2 est en lecture seule pour les membres du groupe)
⇒ ne pourra pas accéder à fic3 (paul n'est pas dans le groupe exp)
Alain (ou root) modifie les droits pour fic3 qui sont à présent :

```
-rw-rw-r-- 1 fic3 alain exp .....................
```

Paul pourra lire fic3, comme tout utilisateur de la machine (lecture pour "other")

5.2 Environnements

5.2.1 Environnement d'accueil de l'utilisateur

La gestion des utilisateurs peut se résumer à la déclaration des deux fichiers vus dans la leçon précédente. D'autres fonctionnalités sont néanmoins utilisées de façon courante, nous verrons les principales.

Les fichiers où sont gérées les caractéristiques communes des utilisateurs.

Le fichier /etc/motd message of the day signifie "mot du jour". Ce fichier contient des informations qui seront affichées à chaque utilisateur lors de son login. Il n'a de limite dans son contenu que votre imagination ... *exemple de contenu*

Listing 5.3– Exemple de fichier /etc/motd

```
**********************************************************************
RAPPEL : aujourd'hui coupure de 12 heures à midi pour maintenance
**********************************************************************
```

ou plus convivial :

```
BONNE ANNEE 2013
```

Le fichier /etc/profile Ce fichier est exécuté systématiquement à chaque login utilisateur. Son contenu peut être le positionnement de variables, l'éxécution de commandes etc ...

Exemple : (on suppose que tout le monde est en ksh)

On souhaite positionner la variable d'environnement $PATH de tous les utilisateurs à **/usr/bin** et **/usr/local/bin** et au répertoire courant (**.**) pour root seulement.

```
1 if [ "$LOGNAME" = root ]
2 then PATH=/usr/bin:/usr/local/bin:.
3 else
4 PATH=/usr/bin:/usr/local/bin
```

autre exemple :

On souhaite modifier le prompt des utilisateurs afin d'y ajouter leur nom et le répertoire courant, sauf pour root pour lequel un prompt particulier doit attirer l'attention

```
1 if [ "$LOGNAME" = root ]
2 then PS1="##"
3 else
4 PS1=$LOGNAME:$PWD"%"
```

Prompt affiché en étant connecté en tant que root :

```
##
```

Prompt affiché en étant connecté en tant que paul :

```
paul:/home/paul%
```

et lorsque paul ira sous un répertoire /commun/fichiers son prompt deviendra [1]

```
paul:/commun/fichiers%
```

1. rappel les variables pre définies s'exportent "toutes seules"

Le fichier où sont gérées les caractéristiques personnelles des utilisateurs. /$HOME/.profile Ce fichier caché [2] se renseigne de la même façon que le fichier /etc/profile commun. Il est cependant unique pour chaque utilisateur et permet donc de renseigner des caractéristiques personnelles.

Attention cependant : ce fichier peut être maintenu par chaque utilisateur, aussi si vous prévoyez par exemple de consigner l'enregistrement de chaque connexion dans un fichier, c'est dans le fichier général que vous le ferez. Le fichier **/etc/profile** s'exécute en effet en premier, puis le **/$HOME/.profile** .

D'autres fichiers sont concernés, que nous verrons dans le chapitre suivant, fichiers qui permettront d'autoriser ou non l'exécution de certaines procédures.

La commande su Cette commande permet de prendre l'identité d'un autre utilisateur de façon provisoire. On vous demandera bien sûr le mot de passe de l'utilisateur dont vous comptez utiliser le compte. Seul root est dispensé de mot de passe, *et peut donc à tout moment prendre l'identité de quelqu'un*.

```
su [-] [logname]
```

L'utilisation du "**-**" indique que les variables d'environnement de l'utilisateur ciblé seront positionnées. Si aucun nom n'est précisé, c'est l'identité de root qui sera prise :

la commande su sans le "-" permet d'exécuter un shell avec un User-ID et un Group-ID différents. Par défaut, la commande su ne change pas de répertoire. Elle positionne les variables d'environnement $HOME et $SHELL à partir des valeurs lues dans le fichier des mots de passe, et si l'utilisateur demandé n'est pas root, renseigne les variables $USER et $LOGNAME. Par défaut le shell exécuté n'est pas un shell de connexion : il n'y a donc pas de modification du prompt. La commande su avec le "-" appelle un shell de connexion. Elle invalide toutes les variables d'environnement (y compris $USER et $LOGNAME) sauf $TERM, $HOME, et $SHELL (qui sont renseignées comme décrit ci-dessus). Elle se déplace dans le répertoire d'accueil de l'utilisateur (contenu dans $HOME). Ainsi, dans l'exemple du cours, la commande su, permet de passer root sans exécuter un shell de connexion, le prompt n'est pas modifié, ni le répertoire de travail. La commande su - alain par contre exécute un shell de connexion. Les variables d'environnement sont repositionnées (prompt y compris) et le répertoire de travail est modifié (passe au répertoire d'accueil de l'utilisateur "alain").

exemple : On suppose que le fichier **/etc/profile** positionne la variable PS1 comme indiqué dans l'exemple ci dessus.

⇒ Je suis paul , mon prompt est `paul:/home/paul%`
je connais le mot de passe de root, j'entre la commande `su`, après saisie du mot de passe, mon prompt devient `root:/home/paul` et j'ai les droits de root.

⇒ Je suis paul, mon prompt est `paul:/home/paul%`
je connais le mot de passe de alain, j'entre la commande `su - alain`

2. notez le point devant son nom

les variables de alain seront positionnées, après saisie du mot de passe mon prompt devient `alain:/home/alain` et j'ai les droits de alain.

5.3 La visualisation des utilisateurs

5.3.1 La commande who

Cette commande liste le nom de login des personnes connectées sur la machine, le numéro de leur écran, la date et l'heure de début de connection.

Listing 5.4– Commande who

```
$ who
dupont   pts/0 Dec 5 15:20
terrieur pts/1 Dec 5 10:54
hisson   pts/2 Dec 5 12:32
andre    pts/3 Dec 5 13:30
```

Les écrans sont pour Unix des fichiers spéciaux qui sont regroupés sous le répertoire `/dev`.

Ainsi, dans l'exemple précédent, les écrans sont des fichiers /dev/pts/0 , /dev/pts/1, /dev/pts/2 et /dev/pts/3

Il est donc possible d'envoyer un message à un utilisateur directement sur son écran en utilisant la commande `echo` .

Listing 5.5– Envoi de message sur écran

```
$ echo Merci de vous deconnecter > /dev/pts/1
```

*affichera **Merci de vous deconnecter** sur l'écran de **terrieur***

5.3.2 La commande finger

Cette commande donne un résultat assez proche de la commande `who`, avec cependant des informations un peu plus précises comme :

⇒ le nom de la personne connectée (issu de la zone commentaire du fichier `/etc/passwd`)

⇒ le nom de la machine depuis laquelle une session est ouverte, qui peut être sur le même réseau local, ou une connection depuis un autre réseau (*cas de Paul Hisson dans l'exemple suivant*)

⇒ le temps d'inactivité (le temps depuis lequel la personne n'a pas soumis une requête depuis son poste)

Listing 5.6– Commande `finger`

```
$ finger
Login    Name                TTY  Idle  Login  Time   Office  Phone
duppont  Albert Duppont   pts/0    5d  Ven    07:52
hisson  Paul Hisson     pts/2    Mar    14:12
```

5.3.3 Environnement des utilisateurs

Les commandes que nous voyons ici peuvent s'exécuter dans les "**profile**" et permettent de gérer des restrictions sur les volumes disques et fichiers accordés à chaque utilisateur.

Les quotas Les quotas permettent de restreindre l'espace disque utilisé par chaque utilisateur.

Plusieurs commandes sont nécessaires pour mettre en oeuvre cette restriction

```
quotacheck -f nom_du_point_de_montage_du_file_system
```

cette commande créée un fichier quotas sous la racine du file système. Si le périphérique **/dev/hda1** est monté sous **/arbo**, il y aura un fichier **/arbo/quotas** qui sera créé.

```
edquota nom_de_l'utilisateur
```

cette commande permet de positionner pour l'utilisateur désigné les quotas sur les files systems auquels il a accès. Les quantités s'expriment en **blocks** (de 1K pour linux) pour la taille autorisée, et en **nombre d'inodes**. Les valeurs "**hard**" sont les maximales à atteindre, les valeurs "**soft**" déclenchent un message d'alerte à l'utilisateur (soft < hard !).

exemple de contenu d'un fichier quota :

```
fs /home blocks (soft = 900, hard = 1024)inodes (soft = 90, hard
 = 102)
```

Dans notre exemple, nous sommes très généreux et n'autorisons que 1 Mo par utilisateur !

```
edquota -p user1 user2
```

On attribue à user2 les mêmes quotas qu'à user1 ...

```
quotaon -a
```

met en place les quotas

`quotaoff`

désactive les quotas

`repquota /rep`

affiche les quotas sur ce file system /rep

Les limites permettent d'éviter qu'une personne monopolise par un programme erroné toutes les ressources de la machine. Les valeurs positionnées sont des maximales.

`ulimit` `option valeur`

cette commande restreint les possibilités d'utilisation des ressources du système.

Les valeurs les plus usitées :

a affiche toutes les valeurs en cours

f taille en blocks d'un fichier

n nombre de fichiers ouverts

t temps processeur

L'utilisateur peut modifier ses propres limites à la baisse, jamais à la hausse

5.4 Préserver les données utilisateurs

5.4.1 Les commandes de sauvegarde

Le stockage d'informations sur un disque n'est pas suffisamment fiable pour espérer s'en contenter. De plus, il est parfois nécessaire de reprendre une ancienne version d'un ou plusieurs fichiers, ou plus simplement de s'échanger des fichiers sous un format compacté. Les commandes que nous voyons ici sont celles que vous pourrez utiliser afin de réaliser vos sauvegardes, ou celles que vous utiliserez lors de l'installation de nouveaux outils.. A noter que les dates des fichiers sauvegardés restent les dates initiales des fichiers.

La copie physique `cpio`

-o lit sur l'entrée standard les noms de fichiers à recopier sur la sortie standard. Cette commande s'utilise donc avec pipes et indirections

-i restaure les fichiers depuis l'entrée standard ; s'utilise aussi avec pipes et indirections

-p utilise l'entrée standard mais cible un répertoire

Exemple : J'ai un répertoire site dans lequel j'ai stoké (entre autres) mes pages web : `10a.htm 11a.htm 12a.htm index.htm img1.gif img2.gif` . Je veux sauvegarder seulement les pages sur une clé USB dans un fichier global appelé save

Listing 5.7– Sauvegarde de fichiers avec cpio

```
$ cd site
$ find . -name '*.htm' -print | cpio -o > /mnt/usb_key/save
```

Sous un répertoire nouveau_site, je veux restaurer mes fichiers

Listing 5.8– Restauration de fichiers avec cpio

```
$ cd nouveau_site
$ cpio -i < /mnt/usb_key/save
```

contenu de nouveau_site après :

```
10a.htm 11a.htm 12a.htm index.htm
```

Je ne veux pas passer par un fichier global, mais directement mettre sur la clé USB les images (.gif) de mon site

Listing 5.9– Sauvegarde des images avec cpio

```
$ cd site
$ find . -name '*.gif' | cpio -p /mnt/usb_key
```

L'archivage `tar`

Cette commande est utilisée pour créer un fichier spécial qui sera lu séquentiellement. La copie d'un répertoire est récursive. Les options :

-x extrait les fichiers depuis le fichier archive

-c créée le fichier archive

-t visualise le contenu de l'archive

-v mode 'verbeux' : la liste des fichiers traités est envoyée à la sortie standard

-f nom du fichier archive

Exemple : Je souhaite sauvegarder tous les fichiers et répertoires de /mes_fichiers dans lequel j'ai deux répertoires rep1 et rep2 qui contiennent respectivement :

Listing 5.10– Contenu du répertoire rep1

```
$ ls * mes_fichiers
rep1 :
fic1 fic2 fic3
rep2 :
fica ficb ficc ficd
```

Listing 5.11– Contenu du répertoire rep2

```
$ cd /mes_fichiers
$ tar -cvf /mnt/usb_key/save_mesfichiers *
rep1/
rep1/fic1
rep1/fic2
rep1/fic3
rep2/
rep2/fica
rep2/ficb
rep2/ficc
rep2/ficd
```

Grâce au mode verbeux, on voit que la commande tar a exploré l'arborescence où je me trouve. La commande tar avec l'option t liste le contenu du répertoire

Listing 5.12– Affichage de la liste des fichiers d'une archive

```
$ tar -tvf /mnt/usb_key/save_mesfichiers
drwxr-xr-x grossin/teachers 0 2001-10-15 13:48:02 ./rep1/
-rw-r--r-- grossin/teachers 7729 2001-10-15 13:06:18 ./rep1/fic1
-rw-r--r-- grossin/teachers 9286 2001-10-15 13:06:18 ./rep1/fic2
-rw-r--r-- grossin/teachers 8750 2001-10-15 13:06:18 ./rep1/fic3
drwxr-xr-x grossin/teachers 0 2001-10-15 13:48:08 ./rep2/
-rw-r--r-- grossin/teachers 382 2001-10-15 13:06:18 ./rep2/fica
-rw-r--r-- grossin/teachers 382 2001-10-15 13:06:18 ./rep2/ficb
-rw-r--r-- grossin/teachers 382 2001-10-15 13:06:18 ./rep2/ficc
-rw-r--r-- grossin/teachers 382 2001-10-15 13:06:18 ./rep2/ficd
```

La recopie sur un répertoire de ces fichiers et répertoires (avec création des repertoires) est faite par la commande

Listing 5.13– Extraction d'une archive

```
$ tar -xf /mnt/usb_key/save_mesfichiers
```

5.5 Exercices

Exercice 8 :

Question 8.1 : Créer un compte pour 4 utilisateurs

Question 8.2 : Donnez à chacun le droit d'aller lire les fichiers des 3 autres mais sans pouvoir les modifier

Question 8.3 : Un des utilisateurs ne souhaite pas communiquer le contenu d'un fichier. Comment va t il faire ? Arrivera-t-il à ses fins ?

Question 8.4 : Ecrire un shell mis à disposition de chacun permettant de sauvegarder ses fichiers sous /repertoire_de_save/nom de la personne

5.6 Correction des exercices

Correction Exercice 8 :

Solution question 8.1 :

création d'un groupe pour mes 4 utilisateurs depuis l'utilisateur root

```
vi /etc/group
```

ligne contenant :

```
group::10:u1,u2,u3,u4
```

Création des 4 utilisateurs

```
vi /etc/passwd
```

lignes contenant :

```
u1::200:10:Utilisateur 1:/home/u1:/bin/ksh
u2::201:10:Utilisateur 2:/home/u2:/bin/ksh
u3::202:10:Utilisateur 3:/home/u3:/bin/ksh
u4::203:10:Utilisateur 4:/home/u4:/bin/ksh
```

Création des répertoires d'accueil

```
$ mkdir /home/u1
$ mkdir /home/u2
$ mkdir /home/u3
$ mkdir /home/u4
```

Appropriation du répertoire par son propriétaire

```
$ chown u1:group /home/u1
$ chown u2:group /home/u2
$ chown u3:group /home/u3
$ chown u4:group /home/u4
```

Solution question 8.2 :

Modification des droits d'accès

```
$ chmod 750 /home/u1
$ chmod 750 /home/u2
$ chmod 750 /home/u3
$ chmod 750 /home/u4
```

Solution question 8.3 :

Interdiction de lire un fichier par les autres : L'utilisateur va modifier les droits sur son fichier en mettant

```
chmod 600 fichier
```

Création du shell

```
vi save
```

lignes contenant :

```
$ A=`whoami`
$ tar -cf /repertoire_de_save/$A /home/$A/*
```

Autorisation d'exécuter pour tous et modification juste par moi

```
chmod 711 save
```

Chapitre 6

Les processus - BH -

Dans ce chapitre, nous présentons les processus à deux niveaux. D'une part, nous montrons comment le système gère ces processus : les structures de données qu'il utilise et les algorithmes. Cette gestion système est abordée dans la première section, mais également tout au long des sections suivantes. D'autre part, nous présentons les commandes utilisateurs relatives aux processus. Nous montrons, par exemple, comment lister l'ensemble des processus s'exécutant sur la machine, comment envoyer un signal, comment exécuter un processus de différentes manières, etc.

6.1 La gestion des processus par le système

6.1.1 Qu'est ce qu'un processus ?

Un programme produit par compilation et édition de liens est un objet inerte correspondant au contenu d'un fichier sur disque.
Un **processus** est un objet dynamique. C'est une abstraction qui correspond à l'exécution d'un programme. Il est identifié par un numéro, le PID (Processus IDentifier).

Un processus est constitué, en plus du programme exécuté (suite d'instructions), de :

- ⇒ l'état du processeur à un instant donné ou contexte d'exécution. Ce contexte d'exécution est composé de l'ensemble des valeurs contenues dans les registres du processeur.
- ⇒ les données du programme ou contexte mémoire. Ce contexte mémoire est l'espace mémoire alloué au processus pour son exécution.

Le système Unix est multi-tâches : plusieurs processus peuvent s'exécuter en même temps. En réalité, à un instant donné, le processeur ne peut exécuter qu'un seul programme. Mais il peut intervenir pour plusieurs programmes en une seconde, ce qui donne à l'utilisateur l'illusion de la simultanéité. Le module système, appelé `ordonnanceur`, est chargé d'allouer le processeur aux différents processus.

6.1.2 Création de processus

Dans Unix, un processus est crée uniquement par un autre processus.

On peut donc définir une `généalogie` des processus ou arborescence de processus. Un processus spécifique, appelé processus `init` est la racine de l'arborescence, chaque processus possède un processus père (sauf le processus `init`) et peut avoir zéro ou plusieurs processus fils.

Un processus est `créé` par duplication du processus père (appel système `fork()`). Le processus fils est une copie du processus père pour la plupart de ses attributs. Les deux processus s'exécutent de façon concurrente. Le processus fils exécute le même programme que son père sur une copie des données de celui-ci lors de l'appel système et donc en particulier une copie de la pile d'exécution.

Les primitives de `recouvrement` permettent à un processus de charger en mémoire un nouveau programme binaire en vue de son exécution. Dans ce cas, le processus fils nouvellement crée n'exécute pas le code de son père mais un autre programme. Les appels systèmes `exec()`, exécutées après un `fork()` permettent de réaliser un recouvrement, il suffit de préciser le programme à exécuter.

Un processus père peut se mettre `en attente` de la terminaison d'un de ses processus fils à l'aide de l'appel système `wait()`. L'exécution du père est suspendue jusqu'à ce qu'un processus fils se termine.

Dans le chapitre suivant, nous verrons comment créer un processus dans un programme C à partir des primitives ci-dessus.

Exemple : Exécution d'une commande dans un shell (figure 6.1).
Quand on tape une commande dans un processus shell, celui-ci crée un processus fils shell qui va exécuter la commande. Pendant l'exécution du processus fils, le processus père est mis en attente. Il reprend son exécution à la fin de l'exécution du processus fils. Ainsi, les primitives suivantes sont exécutées :

- ⇒ primitive `fork()` pour dupliquer le processus shell
- ⇒ primitive `exec()` pour que le processus fils exécute la commande qui est passée en argument de cette primitive.
- ⇒ le processus shell père est mis en attente de la terminaison de son processus fils grâce à la primitive `wait()`.

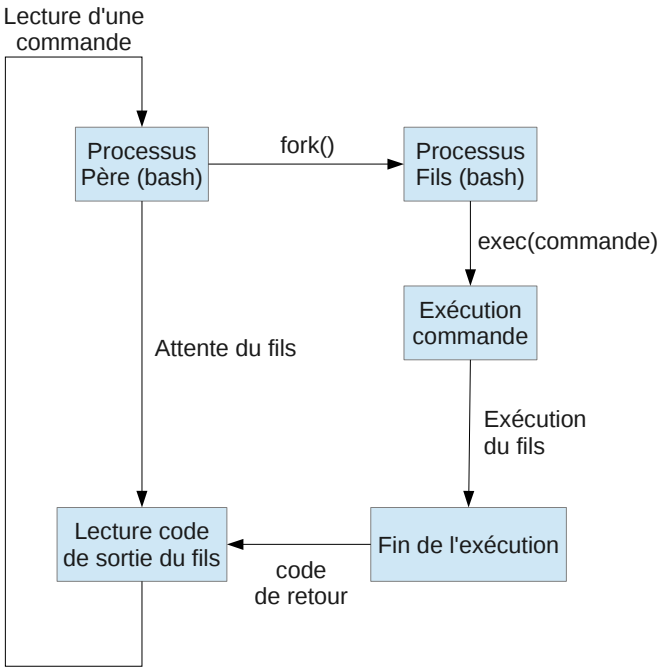

FIGURE 6.1 – Création d'un nouveau shell

Démarrage du système Lors du démarage du système UNIX, le système créé le processus `init`, ce sera le premier processus (PID = 1). Ce processus `init` lit des fichiers de configuration puis lance un ensemble de processus et en particilier des processus chargés de surveiller les terminaux (`getty`). Lorsqu'un utilisateur vient pour utiliser la machine, il se logue sur un terminal. Le processus `getty` associé à ce terminal crée un processus fils, le processus `login`. Ce processus `login` vérifie que l'utilisateur a le droit de se connecter à la machine (à l'aide du fichier `/etc/passwd`) et crée un processus fils shell qui va prendre en charge les commandes tapées par l'utilisateur et pour chacune d'elles créer un processus fils.

6.1.3 Les différents types de processus

Il existe 2 types de processus :

⇒ les processus utilisateurs : ils sont lancés depuis un terminal et sont associés à l'utilisateur qui les a lancés ; ils exécutent des séquences de code noyau uniquement lors de l'utilisation des appels système.

⇒ les processus systèmes : ils s'exécutent en mode noyau (ils ne font pas d'appels système) et ont comme propriétaire l'utilisateur privilégié. A l'initialisation du système, le processus `init` crée plusieurs processus systèmes. Certains proces-

sus sont des processus de premier plan, c'est à dire qu'ils interagissent directement avec l'utilisateur. D'autres sont des processus d'arrière plan, leur rôle est d'assuer un certain nombre de services généraux accessibles à tous les utilisateurs du système, ils sont appelés `démons` (deamon). Par exemple, un démon gère les couriers électroniques entrants, il est activé uniquement lors de l'arrivée d'un message, un démon gère les requêtes sur des pages Web hébergées sur l'ordinateur, un autre les impressions, etc.

6.1.4 Les caractéristiques d'un processus

Pour gérer les processus, et en particulier, l'ordonnancement de leur exécution sur le processeur, le système possèdent des tables où sont stokées des informations sur ces processus.

Unix gère deux structures de données relatives aux processus :

⇒ `La table des processus` (ou structure `proc`) contient des informations sur tous les processus exécutés sur l'ordinateur. Elle possède une entrée par processus et réside en permanence en mémoire. Elle permet entre autre de gérer l'ordonnancement des processus.

⇒ `La structure U` (ou structure User) : elle est associée à un processus. Elle est transférée sur disque lorsque le processus qui lui est associé n'est pas en mémoire.

Informations contenues dans les structures proc et User Un processus est caractérisé dans le système UNIX par les informations suivantes, réparties dans la table des processus et la structure U :

⇒ son identificateur le PID,

⇒ l'identificateur de son processus père le PPID,

⇒ l'identificateur de l'utilisateur qui a lancé son exécution (UID) et du groupe auquel il appartient (GID),

⇒ le répertoire courant (répertoire de travail),

⇒ les fichiers ouverts par ce processus, référencés dans la *table des descripteurs de fichiers*. Cette table possède une entrée par fichier ouvert, les trois premières entrées de cette table correspondent successivement à l'entrée standard, la sortie standard et l'erreur standard du processus (comme nous l'avons déjà vu). Chaque entrée de la table pointe sur la table des inodes.

⇒ le masque de umask donnant les droits d'accès aux fichiers,

⇒ l'état du processus

⇒ les informations pour l'ordonnancement, en particulier la priorité d'exécution du processus *elle est recalculée péridioquement*,

⇒ les temps d'exécution (systeme, utilisateur)

⇒ le terminal de contrôle, qui est le terminal depuis lequel a été lancé le processus.

⇒ les informations sur la gestion mémoire : taille des différents segments constituant le processus, et les pointeurs sur les tables de pages associées.

⇒ un certain nombre de valeurs limites du processus comme la taille maximum des fichiers que peut créer le processus,
⇒ des informations sur les signaux, etc ...

Le système de fichiers /proc On peut accéder à un ensemble d'informations sur les processus en cours d'exécution à l'aide du système de fichiers `proc`. Ce système de fichiers (vu dans le chapitre 2) qui est monté sur le répertoire `/proc` contient, entre autre, un ensemble de répertoires correspondant aux différents processus s'exécutant sur la machine. Chaque répertoire a pour nom le PID du processus décrit. Dans chacun de ces répertoires, on trouve un ensemble de fichiers contenant des informations sur le processus concerné. Par exemple, le fichier `status` contient l'état du processus, le fichier `environ` contient les variables d'environnements du processus, le fichier `cwd` est un lien sur le répertoire courant, le répertoire `fd` contient des liens vers les fichiers ouverts du processus, le fichier `mem` le contenu de l'espace d'adressage, etc ...

6.1.5 L'espace mémoire d'un processus

La mémoire virtuelle

Le `gestionnaire de la mémoire` gère la hiérarchie de la mémoire. Son rôle est de conserver la trace de la partie de la mémoire en cours d'utilisation, d'allouer cette mémoire aux processus qui en ont besoin, de la libérer quand ils ont terminé leur travail, et de gérer le va et vient entre la mémoire principale et le disque quand cette mémoire principale est trop petite pour contenir tous les processus.

La mémoire virtuelle repose sur le principe suivant : la taille mémoire nécessaire pour un processus peut dépasser la capacité disponible de la mémoire physique. Le système conserve les parties de programmes en cours d'utilisation dans la mémoire principale et le reste sur le disque.

Les adresses générées par un programme sont appelées `adresses virtuelles`. L'unité de gestion mémoire ou MMU (Memory Management Unit) intercepte toute instruction manipulant une adresse mémoire pour traduire l'adresse virtuelle en son correspondant réel (mémoire réelle) à partir de tables de pages.

L'espace d'adressage virtuel est découpé en pages logiques toutes de même taille, ce qui autorise à dire qu'une adresse n'est plus un simple indice de 32 bits, mais un numéro de page codé sur 20 bits et un déplacement dans cette page codé sur 12 bits.

En découpant aussi la mémoire réelle en pages de la même taille, on peut charger en mémoire réelle les pages effectivement utilisées au fur et à mesure, et les enlever lorsqu'elles ne sont plus utilisées. La condition nécessaire, est de pouvoir disposer d'un moyen de savoir que les pages de la mémoire virtuelle se trouvent actuellement en mémoire et où elles se trouvent.

L'espace mémoire virtuelle du processus

Tout processus Unix a un espace de mémoire virtuelle constitué de trois segments distincts. Cet espace commence à l'adresse virtuelle 0 comme l'illustre la figure 6.2.

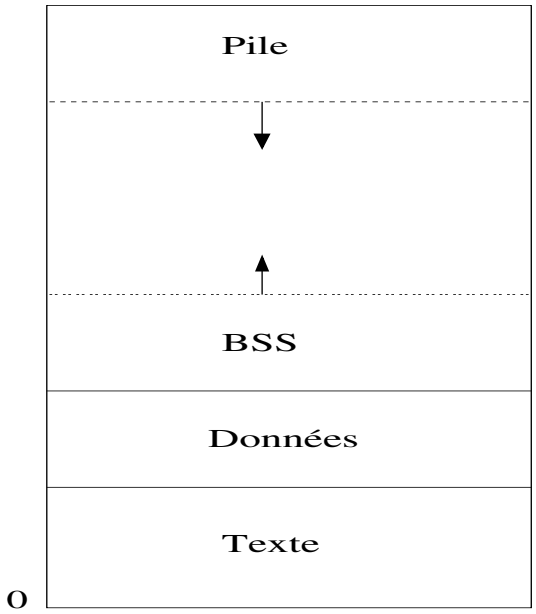

FIGURE 6.2 – Espace mémoire d'un processus

Le segment de texte : contient les instructions en langage machine qui forment le code exécutable. Ce code est produit par le compilateur et l'assembleur qui traduisent à eux deux les programmes écrits en langages plus évolués. En principe, on ne fait que lire le segment de texte. Si deux utilisateurs exécutent le même programme, il est possible que les processus correspondants partagent le même segment de texte. Ceci est géré par les mécanismes de mémoire virtuelle.

Le segment de données : contient l'espace de stockage des variables du programme : chaînes de caractères, tableaux et autres données. Le contenu du segment de données varie ainsi que sa taille.
Le segment de données contient deux parties :
⇒ une partie pour les données initialisées,
⇒ l'autre partie pour les données non initialisées ou BSS : ces données valant 0 il n'est pas utile de les stocker, mais de garder leur place.

Le segment de pile : il commence en haut de l'espace d'adressage virtuel et croit vers les adresses basses. Quand un programme commence son exécution, sa pile n'est pas vide. Elle contient en effet toutes les variables d'environnement (shell) ainsi que la ligne de commande envoyée au shell pour demander l'exécution du programme.

6.1.6 Les processus légers

Un processus peut avoir à exécuter en parallèle plusieurs tâches qui partageront toutefois certaines ressources avec leur père afin de faciliter l'échange de données : ce sont les `threads` ou processus légers. Les threads sont du point de vue Unix des processus à part entière qui ont la particularité de partager toutes les ressources de leur père, mais de disposer de leur propre pile. Nous n'utiliserons pas cette notion dans ce cours, elle sera étudiée en master.

6.2 Le droit d'exécution des processus

6.2.1 Les permissions standards

Le processus s'exécute avec les droits positionnés sur le fichier contenant le programme à exécuter. Un programme qui comporte les droits `-rwxr-xr--` pourra être exécuté par son propriétaire mais aussi par les membres de son groupe.

Lors de son exécution, il aura l'identité et le groupe de la personne qui le lance, ce qui déterminera les droits d'accès aux fichiers .

Listing 6.1– Exemple de droit d'exécution de processus

```
-rwxr-xr-- 1 jacques etudes ..... Mon_Programme
-rw-r----- 1 jacques etudes ....  Le_fichier_de_mon_programme
```

`MonProgramme` peut être exécuté par "jacques" ou par les membres du groupe "etudes". Dans `Mon_Programme` , on accède au fichier "Le_fichier_de_mon_programme" pour modifier des informations. Ce fichier est en ***lecture/écriture*** pour son ***propriétaire*** , en ***lecture seule*** pour le ***groupe*** et non accessible pour les autres.

Regardons l'exécution de ce programme par deux types d'utilisateurs différents (selon leurs droits d'accès aux fichiers) :

1. Je suis ”jacques”, j'exécute `Mon_Programme` : lors de l'exécution j'ai les droits de ”jacques” du groupe ”etudes” : je peux donc lire ***et*** modifier le fichier ”Le_fichier_de_mon_programm .
2. Je suis ”paul” du groupe ”etudes”, j'exécute `Mon_Programme` : je peux le faire. Lors de l'exécution je peux lire ”Le_fichier_de_mon_programme” mais ***pas le modifier :*** nous aurons une erreur à l'exécution.

6.2.2 La substitution d'identité

Si nous reprenons l'exemple précédent :
Pour que ”jacques” puisse permettre à ”paul” d'exécuter `Mon_Programme` correctement,

mais sans lui donner l'autorisation de modifier "Le_fichier_de_mon_programme" dans un autre contexte, il faut que "paul" puisse exécuter `Mon_Programme` *avec les droits de "jacques".*

Afin de positionner ce droit particulier d'exécution, on positionne un droit d'exécution spécial, le droit "**s**" qui signifie "substitution d'identité" (**SUID**). Le positionnement de cette valeur a été expliqué au chapitre 2.

Droits de propriétaire

Un fichier exécutable peut être `setuid` c'est à dire qu'au lieu d'être exécuté avec les droits de l'utilisateur qui le lance, il sera exécuté avec les droits du propriétaire du fichier.

Exemple :
La commande `passwd` permet à chacun de modifier son mot de passe, c'est à dire de modifier le fichier `/etc/passwd`. Ce fichier est en lecture pour tout le monde, mais en écriture par son propriétaire (root) seulement.

Les droits de `/usr/bin/passwd` (le programme) et `/etc/passwd` (le fichier) sont :

```
-rw-r--r-- 1 root sys 6805 Oct 17 15:11 /etc/passwd
-r-s--x--x 1 root sys 13536 Jul 12 2000 /usr/bin/passwd
```

On voit que tout le monde a le doit d'exécuter la commande, mais que lors de l'exécution on prendra les droits du propriétaire de la commande (root) et on pourra donc modifier le fichier `/etc/passwd`*.*

Droits de groupe

Un fichier exécutable peut être `setgid` c'est à dire qu'au lieu d'être exécuté avec les droits de `groupe` de l'utilisateur qui le lance, il sera exécuté avec les droits de `groupe` du propriétaire du fichier.

Exemple :

Je suis `pierre` du groupe `drh`.

J'ai un fichier `MonFichier` qui contient : les noms des employés, leur numéro de téléphone et le montant de leurs revenus annuels.

Seules les personnes appartenant au groupe `drh` (Direction des Ressources Humaines) peuvent visualiser l'intégralité des informations.

J'écris un programme exécutable (pas un shell script) `ListeDesEmployes` qui affiche seulement les noms et numéros de téléphone des employés. Ce programme peut être exécuté par tous les utilisateurs afin de vérifier si les informations sont valides. Par contre, je ne souhaite pas que des personnes n'étant pas dans le groupe `drh` puisse voir les revenus des employés.

`MonFichier` aura les droits : `-rw-r-----` *(je suis le seul à le modifier, mon groupe a le droit de le lire, mais pas les autres)*.

Le programme de listage des informations pour tout un chacun aura les droits : `-rwxr-sr-x`.... ListeDesEmployes

Lorsque `jacques` du groupe`etudes` exécutera `ListeDesEmployes` il appartiendra pendant cette exécution au groupe `drh` et pourra donc lire le fichier.

6.2.3 Le sticky bit

La dernière permission spéciale qui existe, est la permission "**t**" qui remplace le "x" du groupe "other".

Cette permission se rencontre sur les exécutables mais aussi sur les répertoires. Elle a donc deux significations :

Sur le répertoire : le sticky bit protége les fichiers contenus dans le répertoire de la suppression. Il résout le problème suivant : ***comment autoriser tout le monde à ajouter des fichiers dans un répertoire mais en leur interdirsant de supprimer ceux des autres personnes ?***
Le sticky bit est positionné sur un répertoire commun, dans lequel chacun peut mettre ses fichiers. Par contre, seul le propriétaire d'un ficher peut le supprimer (et root bien sûr).

Listing 6.2– Exemple sticky bit positionné

```
$ ls -l repCommun
-rwxrwxrwt 25 robert etudes 34304 feb 2007 repCommun
```

Tout le monde peut ajouter des fichiers dans ce répertoire repCommun, par contre chaque fichier de ce répertoire ne peut être supprimé que par son propriétaire

Sur un exécutable : le sticky bit indique que le code de l'exécutable ne doit pas être déchargé de la mémoire, afin de permettre une exécution ultérieure plus rapide.

6.2.4 En résumé

Un processus possède donc deux propriétaires et deux groupes :

⇒ le propriétaire et le groupe réel (= ceux du user qui a lancé le processus)
⇒ le propriétaire et le groupe effectifs (= ceux positionnés sur l'exécutable)

Dans 90 % des cas, réel = effectif

6.3 L'ordonnancement des processus

Comme nous l'avons déjà dit, le système UNIX est multitâches, il exécute les processus à tour de rôle en temps partagé. Le processeur est alloué au processus pour un quantum de temps, lorsque le processus a écoulé ce temps, ou alors quand il est bloqué sur un évènement, il est suspendu (ses registres sont sauvegardés), mis dans la file des processus en attente et le premier de la file est exécuté. Un processus suspendu est composé de son espace d'adressage (image core) et de son entrée dans la table des processus. Le système utilise un algorithme d'ordonnancement pour gérer l'allocation du processeur basé sur un recalcul périodique de la priorité des processus en fonction de différents critères.

6.3.1 Les priorités

Un processus possède deux priorités qui sont utilisée lors de l'ordonnancement des processus :

sa priorité initiale

Elle a une valeur de -20 à 19.

Plus cette valeur est élevée, moins le processus est prioritaire.

Dans UNIX, la priorité initiale des processus utilisateurs est la plupart du temps égale à **zéro**. Cette valeur peut varier selon les distributions.

Un utilisateur peut agir sur la priorité en la réduisant. Par contre, seul le super utilisateur peut augmenter la priorité en attribuant des valeurs négatives.

Un programme dont la priorité est 19 ne s'exécutera que si rien d'autre ne s'exécute sur la machine ... Un programme à qui le super utilisateur attribue la priorité -20 sera prioritaire sur tout, même les appels système.

La modification de la priorité se fait par la commande `nice` :

```
\shell{nice -15 MonProgramme}
```

La priorité est augmentée car les valeurs négatives sont plus prioritaires. Cette commande est donc exécutée par le super-utilisateur

sa priorité immédiate (PRI)

qui est le résultat du calcul suivant :
`(temps CPU / temps de présence) + priorité initiale + valeur éventuelle de nice (NI)`
Cette priorité immédiate est calculée afin d'éviter qu'un seul processus accapare tout le CPU.

6.3.2 Les différents états d'un processus

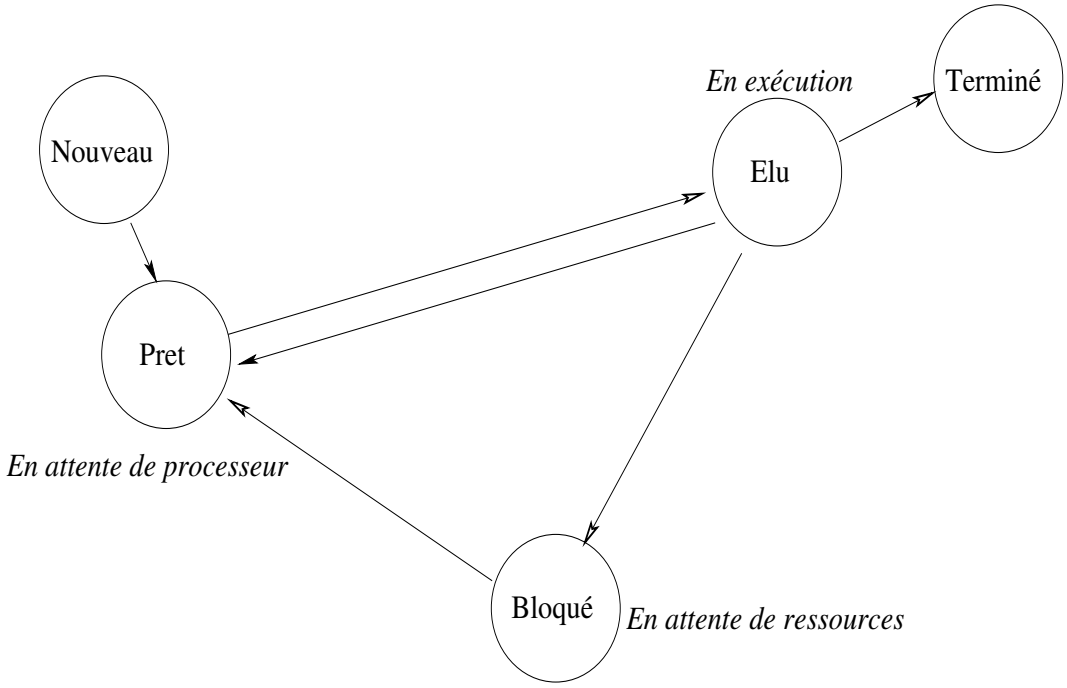

FIGURE 6.3 – Les différents états d'un processus

Quand un processus s'exécute, il est amené à changer d'état. Un processus peut se trouver dans chacun des états suivants (figure 6.3) :

⇒ ***Nouveau*** : le processus est en cours de création.
⇒ ***Elu*** : les instructions sont en cours d'exécution sur le processeur. Le processus quitte cet état soit en passant en état `Bloqué` (attente d'entrée/sortie par exemple), en état `prêt` (suspension du processus par le système dans le cadre de la gestion de l'ordonancement) ou en `Terminé`.
⇒ ***Bloqué*** : le processus attend qu'un évènement se produise comme l'achèvement d'une entrée sortie (par exemple, une impression). A la fin de cette attente, le processus passe dans l'état `Prêt`.
⇒ ***Prêt*** : le processus attend d'être affecté à un processeur (ordonnancement des processus).
⇒ ***Terminé*** : le processus a fini l'exécution.

6.4 Les signaux

6.4.1 La gestion des signaux

Il est parfois nécessaire d'avertir un processus de l'arrivée d'un événement (frappe d'un caractère sur un terminal, violation mémoire, etc). Pour se faire, un utilisateur, un processus ou le système envoie un signal au processus concerné. On peut définir les signaux comme des interruptions logicielles. La prise en compte d'un signal par un processus entraîne l'exécution d'une fonction de gestion du signal appelée `handler de signal`. Sur un système, il existe différents type de signaux, chacun est identifié par une constante standardisée. Ces constantes sont associées à des nombres qui peuvent être **différents** selon les systèmes.

Le tableau (6.1) donne des exemples de signaux : constantes, nombres correspondant en Linux et évènements associés.

Evènement	**Nombres**	**Constantes**
envoyé par un shell pour signaler sa mort	1	SIGHUP
arrêt de la session en cours (équivalent "exit")	3	SIGQUIT
kill du processus (si 15 ne fonctionne pas cas d'une boucle)	9	SIGKILL
terminaison du processus	15	SIGTERM
suspension du processus	19	SIGSTOP
relance du processus précédemment arrêté	18	SIGCONT

TABLE 6.1 – Exemples de signaux

Lorsque le signal est délivré au processus, trois types d'actions peuvent être réalisées :

⇒ ignorer le signal : lorsqu'un signal est ignoré aucune action n'est entreprise au moment da sa prise en compte.

⇒ exécuter l'action par défaut : le noyau associe à chaque signal une action par défaut à réaliser lors de la prise en compte de celui-ci.Ces actions sont au nombre de 5 : abandon du processus, abandon du processus avec création d'un fichier `core`, signal ignoré, processus stoppé, processus reprend son exécution.

⇒ exécuter une fonction spécifique installée par le programmeur : tout processus peut installer un traitement spécifique pour chacun des signaux, ormis le signal `SIGKILL`. Ce traitement spécifique remplace alors le traitemnt par défaut défini par le noyau. Le traitement spécifique peut soit demander d'ignorer le signal, soit définir une action particulière (handler ou gestionnaire du signal).

6.4.2 La commande `kill`

La commande **kill** permet d'envoyer un signal.

Syntaxe :

```
kill -n pid
```

`n` représente le numéro du signal envoyé au processus de numéro `pid`.

Bien que le nom de cette commande puisse faire penser que l'on "tue" le processus, ce n'est pas son unique fonction. Le processus n'est "tué" que lorsqu'il reçoit la commande avec l'option 9 (SIGKILL = signal "kill").

L'option -l de la commande kill liste tous les signaux pouvant être envoyés à un processus :

```
kill -l
```

6.5 La visualisation des processus

La commande **ps** permet d'obtenir des informations sur les processus.

Elle admet un nombre très important d'options, dont nous ne verrons (comme à l'accoutumée) que les plus usitées. Cette commande utilise le système de fichiers `/proc`.

6.5.1 La commande ps

Elle fournit des informations sur les processus actifs du système. Les options permettent d'une part de spécifier des ensembles de processus particuliers et d'autres part, les caractéristiques à visualiser pour chacun d'eux. Sans option, la commande ne traite que les processus associés au terminal dans lequel a été lancée la commande `ps`.

Listing 6.3– La commande ps

```
# ps
PID TTY TIME CMD
15897 pts/1 0:00 ps
24987 pts/1 0:00 bash
```

On voit ici que 2 processus s'exécutent, le processus numéro 24987 qui correspond au shell de ma session, et le processus 15897 qui correspond à la commande "ps" que je suis en train d'exécuter.

6.5.2 La commande ps -ef

L'option -e liste tous les processus en machine (à associer généralement avec un pipe more !). L'option -f spécifie les caractéristiques à afficher :

UID : nom du user qui a lancé le processus

PID : numéro du processus

PPID : numéro du processus père

C : valeur de la priorité

STIME : date (ou heure de lancement si moins de 24 heures)

TTY : terminal depuis lequel la commande a été lancée," ?" si le processus est lancé hors terminal

TIME : temps CPU utilisé depuis le début

CMD : commande et ses paramètres

Listing 6.4– La commande `ps -ef`

```
# ps -ef | more
```

TABLE 6.2 – Résultat de la commande ps -ef

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	Oct22	?	00 :00 :14	init [5]
root	382	1	0	Oct22	?	00 :01 :19	syslogd -m 0
root	392	1	0	Oct22	?	00 :00 :00	klogd
rpc	407	1	0	Oct22	?	00 :00 :00	portmap
nobody	433	1	0	Oct22	?	00 :00 :00	identd -e -o
root	13233	13232	0	09 :29	pts/0	00 :00 :00	login – grossin
duppont	13235	13233	0	09 :29	pts/0	00 :00 :00	-bash

On voit dans cet exemple que le processus père du shell de duppont (PID 13235) est le processus 13233 qui appartient à root et qui a été lancé depuis le terminal "0"

6.5.3 La commande ps -aux

L'option `-a` permet d'afficher des informations sur des processus appartenant à d'autres utilisateurs. L'option `-x` permet d'afficher des informations sur des processus qui n'ont pas de terminal de contrôle. L'option `-u` spécifie les informations à afficher. Elles permettent, entre autre, de repérer les processus les plus "gourmands" en terme de ressources.

Les informations nouvelles affichées avec cette option sont :
⇒ % **CPU** est le temps CPU utilisé / temps écoulé depuis le lancement
⇒ %**MEM** est le pourcentage de mémoire virtuelle utilisée
⇒ **VSZ** est la taille du programme en mémoire (en Ko)
⇒ **RSS** est la taille RAM utilisée (en Ko)
⇒ **STAT** est l'état du processus détaillé dans le tableau (6.3).

R	prêt à s'exécuter
T	Stoppé
S	dormant depuis moins de 20 secondes
I	dormant depuis plus de 20 secondes
Z	terminé , en attente que son processus père effectue un appel : processus zombie
W	swappé
O	processus "orphelin" : le processus père n'exite plus
N	indique que la priorité a été augmenté
<	indique que la priorité a été diminuée

TABLE 6.3 – Valeurs du champs STAT

Comme toujours, cette liste n'est pas exhaustive. Elle est donnée à titre informatif, sachant que ces informations sont utilisées dans l'administration d'un système Unix, pas dans l'utilisation courante.

6.6 L'exécution des processus

Jusqu' à présent, nous avons vu comment lancer un processus depuis sa console, une fois connecté par une session utilisateur sur le serveur. Nous allons voir dans cette leçon qu'il est possible d'exécuter de façon **asynchrone** voire **différée** les processus.

6.6.1 L'exécution asynchrone

Lors de la soummission de commandes depuis une console, le système ne rend la main que lorsque la commande est terminée. Lorsqu'il s'agit d'un simple "ls", cela est assez rapide, mais si vous écrivez une procédure shell qui renomme tous les fichiers d'une arborescence importante en "leur_nom.à_jeter" et si ces fichiers n'ont pas été consultés depuis plus de 3 mois, vous risquez d'attendre longtemps avant de récupérer le prompt et de pouvoir entrer une nouvelle commande.

Par défaut, une commande est excécutée en **premier plan** ou **forground**, c'est à dire de façon intéractive.

La commande &

Pour exécuter une commande en **arrière plan** ou mode **background** (en asynchrone), il faut ajouter le caractère **&** à la fin de la commande, ce qui a pour effet de redonner la main avec l'affichage du `prompt`. Lors de l'exécution d'une commande en background, il est intéressant de redigirer les sorties standard et erreur de la commande dans un fichier pour qu'elles ne s'intercalent pas avec celles de la commande en forground.

Listing 6.5– Commande en mode forground

```
# monProgramme
```

Cette commande s'exécute en mode forground, vous n'aurez le prompt qu'à l'issue de monProgramme

Listing 6.6– Commande en mode background

```
# monProgramme &
%[1]   7106
#
```

Cette commande s'exécute en mode background, vous avez le prompt qui s'affiche à la suite. Pour visualiser le pid du processus lancé, vous consultez la liste de vos processus.

A l'issue de l'exécution, un message à l'écran vous informe :

```
[1]+   Done monProgramme
```

Suspension-Redémarrage de processus

Il est possible de suspendre une commande en cours d'exécution en tapant `ctrl-z` sur le clavier. Ceci retourne un numéro qui pourra être utilisé pour redémarrer le processus en arrière plan `bg` ou premier plan `fg`.

Listing 6.7– Commande `sleep`

```
# sleep 10
^z
[1]+  Stopped                 sleep 10
# sleep 5
^z
[2]+  Stopped                 sleep 5
# fg 1
sleep 10
# bg 2
[2]+ sleep 5&
#
```

La commande `sleep` permet d'endormir un procesus, elle est suspendue par la commande `ctrl-z`, on reprend la main, une autre commande `sleep` est lancée et également suspendue. Puis ces deux commandes sont relancées la première en premier plan et la seconde en second plan.

Exécution asynchrone et terminaison de session

Une exécution en background peut parfois durer plusieurs heures , voire plusieurs jours ! Or, dès que vous quittez votre session, un signal est envoyé à vos processus en cours afin de les tuer. Vous êtes néanmoins prévenu, puisqu'à la commande `exit`, un message vous indique que des processus sont encore en exécution.

Pour lancer un processus en background et pouvoir fermer sa session alors que le processus continue de s'exécuter, il faut préfixer le lancement de la commande par l'instruction `nohup` qui permet d'ignorer les signaux `SIGHUP` (coupure de ligne) et `SIGQUIT` (fermeture de session). La sortie standard et la sortie des erreurs sont redirigées dans le fichier `$HOME/nohup.out`

Listing 6.8– Commande `nohup`

```
# nohup mon_programme &
```

mon_programme est lancé, et continuera de s'exécuter une fois ma session fermée.

6.6.2 L'exécution différée

L'exécution à heure précise

La commande `at` se déclare de façon interactive. Elle se complète avec la date et l'heure souhaitée pour l'exécution. Au prompt `at,` renseignez la commande que vous souhaitez exécuter. La fin de l'entrée se termine par `CTRL-D`.

Listing 6.9– Commande `at`

```
# at 15:00 dec 25
at > mon_programme_de_noel
at > CTRL-D
```

Pour **visualiser** les exécutions en attente, on utilise la commande **atq**.

Listing 6.10– Commande `atq`

```
# atq
8 2001-12-25 15:00 a durant
```

Le '8' indique le numéro de votre job. La date et l'heure suivent ainsi que la file d'attente (a) et le nom de l'utilisateur (durant)

Pour **annuler** l'exécution différée, on utilise la commande `atrm` suivie du numéro de job concerné.

Listing 6.11– Commande `atrm`

```
# atrm 8
```

Le 8 sort de la file d'attente.

L'exécution en séquence : batch

Elle fonctionne de la même façon que la commande `at`, mais elle met simplement l'exécution au bout de la file d'attente. La commande s'exécutera lorque les autres se seront terminées.

Listing 6.12– Commande `batch`

```
# batch
at> mon_job
at> CTRL-D
```

L'exécution cyclique : crontab

La commande `crontab` permet de soumettre de façon cyclique un même job.

L'utilisateur crée un fichier dans lequel seront indiqués les jobs à exécuter de façon cyclique accompagnés des informations nécessaires à leur exécution. Chaque ligne de ce fichier est codifiée de la façon suivante :

⇒ 5 zones (tableau 6.4) qui correspondent chacune à une fréquence. La valeur $\star$ signifie "toutes les valeurs sont autorisées".

zone 1 :	les minutes	valeur : de 0 à 59
zone 2 :	les heures	valeur : de 0 à 23
zone 3 :	le jour du mois	valeur : de 1 à 31
zone 4 :	le mois	valeur : de 1 à 12
zone 5 :	le jour de la semaine	valeur : de 0 à 6 (0 = dimanche)

TABLE 6.4 – Zones du fichier cron

Chaque zone est codifiée par un chiffre, ou une plage de valeurs séparées par un '-' , ou une liste de valeurs séparées par une ','

⇒ la commande à exécuter en fin de ligne

Exemple 1 :
Tous les mardis et jeudis matins à 7 :15 , je veux exécuter le fichier "mon_job" :
Le contenu du fichier "cron" sera :

```
15 7 * * 2,4 mon_job
```

Exemple 2 :
Du 1 au 10 du mois de janvier à 9 heures , je veux exécuter la commande "bonne_annee"
Le contenu du fichier "cron" sera :

```
0 9 1-10 1 * bonne_annee
```

La soumission du fichier renseigné se fait avec la commande : `crontab nomDuFichierCron`.

L'exécution de la commande `crontab fic1` induit le stockage du fichier `fic1` dans le fichier `/var/spoll/cron/mon_logon` où `mon_logon` est le login qui se trouve dans le fichier `/etc/passwd`. Pour chaque `login` dans `/etc/passwd`, si un cron est mis en place, il y aura un fichier dans le répertoire `/var/spool/cron`.

La commande `crontab -e` ouvre directement le fichier `/var/spoll/cron/mon_logon` de l'utilisateur courant à l'aide de l'éditeur de texte spécifié dans sa variable d'environnement `$EDITOR`.

La commande `crontab -l` affiche le contenu du fichier `cron` à l'écran.

6.7 Exercices

Exercice 9 : Ecriture de commande : visualisation des processus

Question 9.1 : Ecrire une commande qui affiche les processus exécutés par l'utilisateur "Dupond" qui a pour uid "501".

Question 9.2 : Ecrire une commande qui affiche le nombre de processus exécutés par l'utilisateur "Dupond" qui a pour uid "501".

Exercice 10 : Ecriture du shell script `execProc`

Question 10.1 : Ecrire un shell script `execProc` qui :

⇒ affiche le nom du shell script
⇒ affiche le PID du shell script
⇒ affiche le PID du processus père du shell script
⇒ affiche le contenu de la variable `PS1`
⇒ affiche le contenu de la variable `varA`
⇒ affiche le contenu de la variable `varB`

Question 10.2 :

⇒ Afficher le PID du shell courant,
⇒ affecter une valeur à chacune des variables `varA` et `varB`,
⇒ affecter la valeur "Bonjour" à la variable d'environnement `PS1`
⇒ exporter la variable `varB`
⇒ lancer le shell script `execProc` des trois manière suivantes :

1. `bash execProc`
2. `execProc`
3. `. execProc`

Commenter les résultats obtenus à partir de ce que vous avez appris dans ce chapitre.

Exercice 11 : Ecriture du shell script `q1.sh`

Ecrire un shell script (`q1.sh`) qui fait une boucle infinie en attendant une entrée clavier et en imprimant cette entrée sur la sortie standard précédée de la date et de l'heure.

Le format de la date et heure doit permettre de faire correspondre l'ordre chronologique à l'ordre alphabétique.

Indications :

⇒ Nous rappelons que la lecture sur l'entrée standard se fait avec la commande `read` de la façon suivante :
`read X` *lit l'entrée standard et la stocke dans la variable $X.*

⇒ Lors de l'exécution de ce programme, pour sortir il faudra entrer `CTRL_C` sur votre clavier, car une boucle infinie, par définition, ne se termine jamais.

Exercice 12 :

Pour ouvrir plusieurs sessions sur votre ordinateur, appuyez sur `ALT+Fn` , n valant de 1 à 6. Vous pouvez donc ouvrir 6 sessions au maximum. Pour passer de l'une à l'autre, vous reprenez les touches `ALT-Fn`. Votre 1ère session est ouverte au boot, vous ouvrirez donc une seconde sesson par `ALT-F2` puis une troisème par `ALT-F3`.

On veut que dans deux sessions différentes les entrées saisies dans l'une apparaissent dans l'autre préfixées par la date et l'heure.

A l'aide du script précédent (`q1.sh`) que faut-il faire ?

Parmi les différentes solutions on exprimera celle qui utilise un fichier intermédiaire contenant l'historique des saisies.

Exercice 13 :

On modifie le script `q1.sh` pour imprimer deux messages toutes les 4 secondes sur la sortie standard à l'aide d'une boucle infinie.

Les messages sont de la forme :
Premier message : `Bonjour processus` *`numero du processus`*
Deuxieme message : `Au revoir processus` *`numero du processus`*

Indications :
Utiliser la fonction `sleep` pour l'attente toutes les 4 secondes.

Exercice 14 :

Dans trois sessions distinctes lancer le script précédent redirigé vers le fichier `sortie` en mode ajout .

Dans une quatrième session, lire le fichier `sortie`.

Que se passe-t-il ?

Est-ce que l'ordre des messages est conservé ?

Rappel :
On change de session par alt+Fn ou n vaut 1 à 6.

Exercice 15 :
Ecrire un shell script qui effectue une boucle infinie. Exécuter ce shell script en arrière plan. Arrêter l'exécution de ce shell script.

Exercice 16 :
Question 16.1 : Ecrire un shell script qui affiche les processus ayant une taille mémoire supérieure à 1500 Ko.
Indication : Ecrire une solution avec la commande `set` et une autre en utilisant les tableaux.

Question 16.2 : Ecrire un shell script qui affiche le nombre de processus ayant une taille mémoire supérieure à 1500 Ko.

6.8 Correction des exercices

Correction Exercice 9 : Commande
Solution question 9.1 :

```
ps -all | grep "501"
```

affichera uniquement les lignes sur lesquelles apparaît l'uid de Dupond, donc le nombre de processus exécutés par dupond. Pour être plus précis il ne faudrait afficher que les lignes dont l'uid de dupont apparait en premier champs du résultat de la commande ps. Voir dernier exercice

Solution question 9.2 :

```
ps -ef | grep "501"| wc -l
```

Correction Exercice 10 : `execProc`
Solution question 10.1 : Le shell script `execProc`

```
1 echo "Mon nom est : $0"
2 echo "Mon PID est : $$"
3 echo "Le PID de mon père est : $PPID"
4 echo "Contenu de PS1 : $PS1"
5 echo "Contenu de varA : $varA"
6 echo "Contenu de varB : $varB"
```

Solution question 10.2 : Avant d'exécuter le shell script, initialisation des variables, affichage du PID du shell courant :

```
1 varA=1
2 varB=2
3 PS1="Bonjour"
4 export varB
5 echo $$
```

1. `bash execProc` un nouveau processus de type shell est créé. On n'a pas besoin de donner les droits d'exécution au shell script. Comme il y a création d'un nouveau processus, seules les variables exportées sont connues du processus fils (varB). La variable `PS1` est exportée (car c'est une variable d'environnement) mais comme il y a création d'un nouveau processus shell, il y a exécution des fichiers d'initialisation qui réinitialisent la variable `PS1`.
2. `execProc` le shell script est exécuté comme une commande, il doit donc avoir les droits d'exécution :

   `chmod u+x execProc`

   Le "." doit être indiqué dans la variable d'environnement `PATH`, sinon on doit donner le chemin absolu de la commande.
   Comme il y a création d'un nouveau processus, seules les variables exportées sont connues du processus fils (varB). La variable `PS1` est exportée (car c'est une variable d'environnement) mais comme il y a création d'un nouveau processus shell, il y a exécution des fichiers d'initialisation qui réinitialisent la variable `PS1`.
3. `. execProc` il n'y a pas de nouveau processus de créé. Le shell script est interprété dans le shell courant. Comme il n'y a pas création de nouveau processus, les variables du shell courant sont connues dans le shell script. Attention : on peut alors modifier ces variables.

Correction Exercice 11 :

```
1 while [ 1 -eq 1 ]
2 do
3    read X
4    echo $(date +%Y/%m/%d-%H:%M:%S)  $X
5 done
```

Correction Exercice 12 :

```
1 Session 1
2 ./q1.sh >> sortie
3
4 Session 2
5 tail -f sortie
```

Correction Exercice 13 :

```
1 while [ 1 -eq 1 ]
2 do
3   sleep 4
4   echo $(date +%Y/%m/%d-%H:%M:%S) Bonjour Processus $$
5   echo $date +%Y/%m/%d-%H:%M:%S) Au revoir Processus $$
6 done
```

Correction Exercice 14 :

Les messages s'affichent chronologiquement.

Correction Exercice 15 :

Exemple de shell script `exBoucle` réalisant une boucle infinie :

```
1 while true
2 do
3 echo "bonjour" > tmp
4 done
```

Exécuter ce shell script en arrière plan :

`exBoucle&`

Arrêter son exécution :

`kill -9 numPid`

En remplaçant `numPid` par l'identificateur rendu par l'exécution en arrière plan.

Correction Exercice 16 :

Solution question 16.1 : Solution avec un tableau :

```
1 i=1
2 ps -all |  while read line
3 do
4   if [ $i -ne 1 ]
5   then
6     tab=($line)
7     if [ ${tab[7]} -ge 1500000 ]
8       then echo $line
9     fi
10  fi
11  i=$((i+1))
12 done
```

Solution avec la commande `set` :

```
1 i=1
2 ps -all |  while read line
3 do
4   if [ $i -ne 1 ]
5   then
6     set $line
7     if [ $8 -ge 1500000 ]
8       then echo $line
9     fi
10  fi
11  i=$((i+1))
12 done
```

Remarque : on ne peut pas utiliser la commande `cut` pour récupérer les différents champs du résultat de la commande `ps` car le nombre d'espaces entre chaque champs n'est pas fixe.

Solution question 16.2 :

```
1 i=1
2 ps -all |  while read line
3 do
4   if [ $i -ne 1 ]
5   then
6     tab=($line)
7     if [ ${tab[7]} -ge 1500000 ]
8       then echo $line >> nbProc
9     fi
10  fi
11  i=$((i+1))
12 done
13 echo "$(wc -l nbProc)"
```

Chapitre 7

Réseau -GL-

7.1 Introduction

Les réseaux sont devenus omniprésents dans notre monde actuel, tant au niveau du cercle privé ("box" reliée à quelques machines personnelles) que mondial (Internet, téléphonie...)

Pour toute communication il est nécessaire de définir un protocole : D'abords purement privés, ceux-ci se sont progressivement répandus pour donner naissance aux solution réseaux actuelles. Dans cette partie, nous allons présenter ces protocoles.

7.2 Les protocoles réseau

Les plus anciens protocoles réseaux ont souvent été développés par des éditeurs et des constructeurs de matériel, à l'époque de l'émergence des premiers réseaux modernes. Ces protocoles, pour la plupart propriétaires, étaient alors fortement liés à une gamme de matériel ou une technologie spécifique et fondentalement incompatibles entre eux, à la fois pour des raisons technologiques (normalisation des signaux, des connectiques, principes de fonctionnement) qu'industrielles (intérêts économiques, brevets, concurrence.)

Après cette première période d'intense fragmentation des technologies, il est rapidement apparu nécessaire de proposer des solutions plus unifiées pour simplifier la compréhension et l'interconnexion entre ces premières approches.

Le modèle OSI, élaboré dans les années 80, représente la première tentative d'établir un modèle commun pour décrire ces différentes implémentations. Ce modèle repose sur une division des traitements à effectuer en 7 couches spécialisées s'occupant chacune d'un aspect particulier de la communication réseau (gestion du matériel, de l'adressage,du transport).

Bien que peu représenté dans les implémentations pratiques, cette organisation en couche se retrouve dans l'immense majorité des protocoles actuels, et le modèle OSI continue à servir de point de comparaison entre technologies distantes.

Le protocole réseau le plus répandu à l'heure actuel est TCP/IP, notamment connu pour son rôle dans la mise en place d'Internet et sa présence dans un grand nombre de solutions grand public comme ethernet ou les technologies sans fil (Wifi, Bluetooth).

Le protocole TCP/IP lui-même ne fournit que les service associées aux couches 3 à 5 du modèle OSI, les autres couches étant souvent implémentées au niveau matériel (cartes réseaux) ou utilisateur (navigateur Internet, serveur de fichier).

Le schéma ci-après illustre ce parallèle entre les deux modèles.

<table>
<tr><th>OSI</th><th colspan="4">TCP/IP</th></tr>
<tr><td>7 - Application</td><td rowspan="2">FTP</td><td rowspan="2">POP3</td><td rowspan="2">IMAP</td><td rowspan="2">HTTP</td></tr>
<tr><td>6 - Présentation</td></tr>
<tr><td rowspan="2">5 - Session</td><td colspan="2" rowspan="2">TCP</td><td colspan="2" rowspan="2">UDP</td></tr>
<tr><td>4 - Transport</td></tr>
<tr><td>3 - Réseau</td><td colspan="4">IP ICMP</td></tr>
<tr><td>2 - Liaison</td><td colspan="4" rowspan="2">ARP</td></tr>
<tr><td>1 - Physique</td></tr>
</table>

FIGURE 7.1 – Modèles OSI et TCP/IP confrontés

7.2.1 OSI

Nous présentons ici les principales notions concernant les couches OSI 1 à 3 d'un réseau Ethernet (technologie la plus employée pour les réseaux locaux). Ce ne sont ici que les informations principales, chaque couche pouvant à elle seule être le sujet d'un chapitre.

La couche physique

correspond au matériel et au support de transmission. Ce dernier sera fonction de la portée de communication souhaitée, du débit attendu, des coûts et de l'environne-

ment : Il pourra s'agir de fils de cuivre, de fibre optiques, d'ondes hertziennes, ou de tout autre média permettant de véhiculer une information.

La couche liaison

gère le codage de l'information dans un format exploitable par la couche physique, et les informations nécessaires au fonctionnement de celle-ci. Cette couche reste de très bas niveau, sans aucune notion de filtrage, renvoi, ou gestion de priorité.

TABLE 7.1 – Format d'une trame Ethernet

Préambule	SFD	Adr. destination	Adr. source	Type	Données	FCS
7 octets	1 octet	6 octets	6 octets	2 octets	46 à 1500 octets	4 octets

Préambule permet la synchronisation des horloges

SFD indicateur de début de trame, marque la fin du préambule

Adresse source et destination correspond aux adresses MAC (Media Access Control) de la source et de la destination. Ces adresses physiques *uniques* sont associées lors de la fabrication à chaque carte ethernet par le constructeur, indiqué le plus souvent par la valeur des trois premiers octets de l'adresse : Par exemple, une adresse commençant 08 :00 :20 indiquera une carte Sun.

Type type ou longueur des données

FCS somme de contrôle (permet la correction d'erreurs)

La couche réseau

s'occupe de gérer l'adressage des machines, notamment de l'émetteur et du récepteur, tel que nous le détaillerons dans la leçon suivante.

7.2.2 TCP/IP

TCP/IP est un *ensemble de protocoles* permettant d'interconnecter des réseaux hétérogènes à la fois en terme de matériel et de systèmes d'exploitation (OS, Operating System en anglais).

Dès les années 70 le Departement of Defense (DoD = département de la défense américain) développe un modèle réseau basé sur 4 couches afin d'interconnecter ses machines. Ce modèle, toujours d'actualité, est intimement lié à Unix puisque la première mise en oeuvre de TCP/IP a été faite réalisée avec ce système : TCP/IP est présent sur *tous* l'ensemble des variantes Unix modernes.

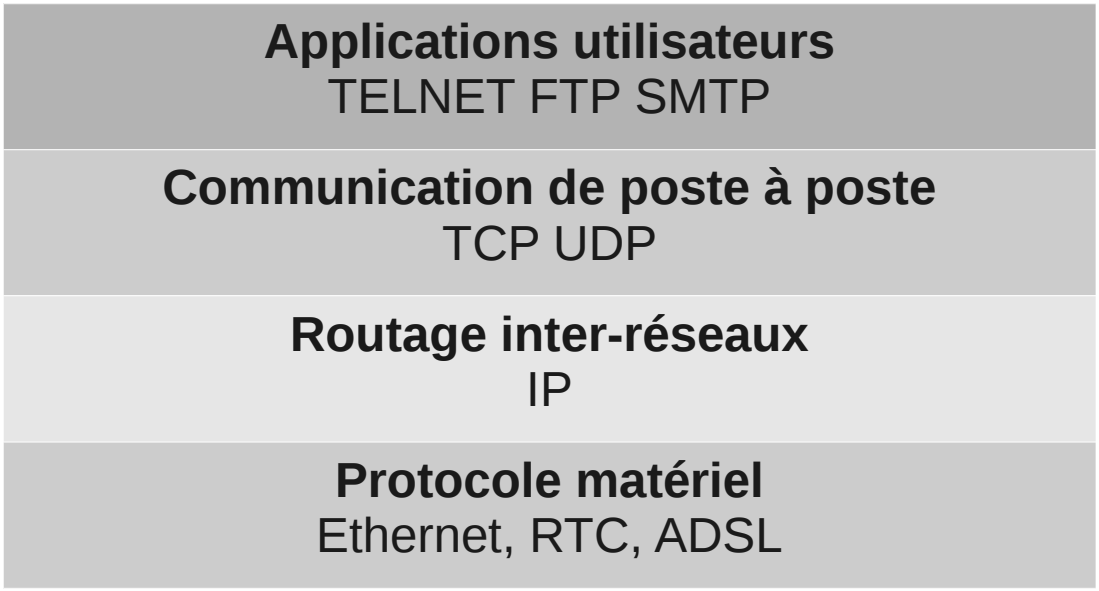

FIGURE 7.2 – Le modèle 4 couches Dod –> TCP/IP

Chacune des "couches" gère un aspect de la communication entre deux machines

1. La couche la plus basse (accès réseau matériel) comporte les protocoles permettant l'envoi d'informations sur le support physique du réseau (votre ligne téléphonique dans le cas d'un ordinateur relié par modem, par exemple, ou n'importe quel réseau local Token Ring, Ethernet, X25, SNA, FDDI ...)
2. Le routage inter réseau est composé de protocoles permettant d'échanger des données sur des réseaux physiques différents ; leprincipal est "IP", acronyme de Protocole (protocol) d'Interconnexion (inter) de Réseaux (net) = Internet Protocol.
3. La communication de poste à poste est assurée par 2 protocoles qui s'excluent mutuellement :

   *TCP (**T**ransport **C**ontrol **P**rotocol) et UDP (**U**nreliable **D**atagram **P**rotocol)*

   UDP et TCP correspondent à la couche *transport* du modèle OSI.

   La principale caractéristique de cette communication est qu'elle se fait "d'**égal à égal**" (*peer-to-peer*). Chaque ordinateur a le même "poids" que celui avec lequel il communique. Il n'y a pas d'ordinateur "maître" ni d'ordinateur "esclave" dans la communication. Il apparaît néanmoins une notion de client-serveur : l'ordinateur qui envoie une requête est client, celui qui répond est serveur. Cette notion n'est valable que pour le temps d'un échange.
4. La couche application comporte les protocoles qui permettent d'accéder aux informations comme le transfert de fichier (FTP : File Transfert Protocol) , le mail (SMTP : Simple Mail Transfert Protocol et POP3 : Post Office Protocol version 3) ...

Le principe général est donc d'assurer aux applications l'**indépendance par rapport aux réseaux physiques** et de donner les **services qui permettent de communiquer**.

> *A l'heure actuelle, tous les fournisseurs proposent TCP/IP comme protocole réseau. TCP/IP fonctionne donc sur n'importe quelle machine, du PC au mainframe, sur n'importe quel type de réseau physique, réseaux locaux ou lignes séries. Il est fourni avec des Operating System aussi différents que DOS, Windows, OS/2, MVS ... et toutes les déclinaisons d'Unix.*

7.2.3 Les principaux protocoles utilisés dans les couches TCP/IP

Le niveau Inter Reseau : juste au dessus de la couche physique

IP Ce protocole permet de *transporter des données d'un bout à l'autre du réseau* sans se soucier des aspects physiques des réseaux traversés. Grace à IP, il est possible de passer d'un réseau à un autre par l'intermédiaire de passerelles dont le rôle est de permettre de passer d'un réseau à un autre. Une passerelle fait forcément partie d'au moins de deux réseaux, et ne se distingue d'une autre machine que par cette fonction particulière de liaison de 2 réseaux.

En IP (Internet Protocol), passer d'un réseau à un autre se dit *router*. Chaque machine possède une adresse unique grâce à laquelle elle est repérée sur le réseau à laquelle elle appartient.

Les données sont encapsulées dans des paquets constitués d'une part d'un entête, qui précise les informations nécessaires à la remise des données au destinataire, et d'autre part les données elles-même, encapsulées sans transformation.

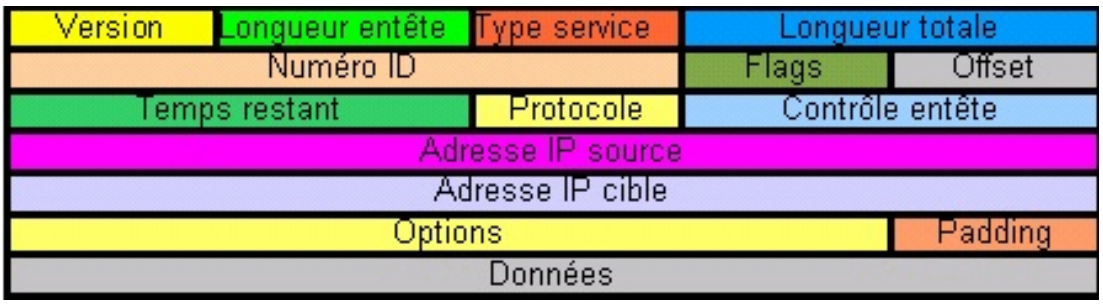

FIGURE 7.3 – Structure de l'entête IP

Version version de protocole utilisé (permet de savoir si on est en IPv4 ou IPv6)

Longueur de l'entête taille de l'en-tête exprimée en mots de 32 bits, nécessaire pour connaître l'adresse de début des données

Type de service Il se décompose en 8 bits dont les 3 premiers indiquent un niveau de priorité, les 3 suivants codifient des drapeaux qui indiquent que l'on souhaite un délai de transmission court (1er bit D "delay"), un haut taux de transfert (2nd

bit T "transfert"), une fiabilité haute (3ème bit R "reliability"). Les 2 derniers bits ne sont pas utilisés. *Le type de service est presque toujours ignoré par les routeurs.*

Longueur du datagramme entête + données codé sur 16 bits. La longueur maximale d'un paquet est donc de 2^{16} - 1 soit 65535 octets

Numéro d'ID identifie le paquet de façon unique : permet de gérer les éventuels doublons générés par le réseau et d'assembler les paquets fragmentés.

Flag drapeaux sur 3 bits indiquant éventuellement les fragments suivants

Offset valeur sur 13 bits indiquant la position du fragment par rapport au paquet d'origine

Temps restant nombre de routeurs que le datagramme peut franchir : décrémenté par chaque routeur à son passage.

Protocole transporté identifie la couche supérieure à qui le paquet est destiné.

Contrôle de l'entête permet de vérifier l'intégrité du paquet.

Options sert aux administrateurs réseaux pour vérifier ou analyser le réseau

Padding ensemble de bits à 0 qui permettent d'aligner l'entête sur un multiple de 32 bits, les options pouvant être positionnés ou non.

** **ICMP** ** Internet Control Message Protocole : ce protocole est dédié au *contrôle de l'état du réseau*, et permet notamment de diagnostiquer les problèmes de routage, de durée de vie des paquets ou de congestion. Il se situe est au niveau liaison, juste au dessus du protocole IP.

Ce protocole est le plus souvent exploité par les routeurs ou par les administrateurs pour obtenir des informations sur le réseau en général, et implémente notamment les fonctions utilisées par des commandes comme `ping` et `traceroute`, qui travaillent avec des paquets ICMP.

** **IGMP** ** Internet Gateway Multicast Protocol : permet d'envoyer des messages à des groupes d'ordinateurs donnés d'un même réseau, par opposition à TCP qui ne gère que la communication point à point ou la diffusion à l'ensemble des machines connectées.

** **ARP** ** Adress Resolution Protocol : S'occupe de la résolution des adresses IP logiques en adresses physiques, dépendantes du média, et employées pour la communication à proprement parler.

Pour ce faire, une requête de type "broadcast", c'est à dire s'adressant à toutes les machines du réseau, est émise par la machine désirant obtenir l'adresse physique associée à une adresse IP donnée, contenant sa propre adresse physique et l'adresse IP recherchée. Si un machine possède cette adresse IP, elle enverra alors une réponse

contenant sa propre adresse physique au demandeur, à l'aide de l'adresse spécifiée dans la requête ARP.

Pour éviter de surcharger le réseau avec des demandes incessantes, le service ARP gère une table de correspondance entre adresses logiques et matérielles : Si le correspondant est déjà dans la table, ARP utilise directement l'adresse enregistrée, et n'envoie une requête broadcast à tous les ordinateurs du réseau que si l'information n'est pas déjà disponible dans la table.

Le contenu de cette table est régulièrement nettoyé, et des nouvelles requêtes sont ainsi parfois envoyés, pour vérifier que les informations sont toujours à jour. La commande unix `arp` permet de visualiser cette table.

Le niveau communication poste à poste

Deux protocoles principaux sont employés à ce niveau :

TCP (Transmission Control Protocol : transmission en mode connecté) : L'analogie couramment employée pour illustrer ce mode est celle d'une communication téléphonique : *la liaison est maintenue* pendant l'ensemble de la communication et des informations entre source et destination afin de *fiabiliser* l'échange. *TCP assure l'intégrité et le séquencement (récupération dans le bon ordre) des données.*

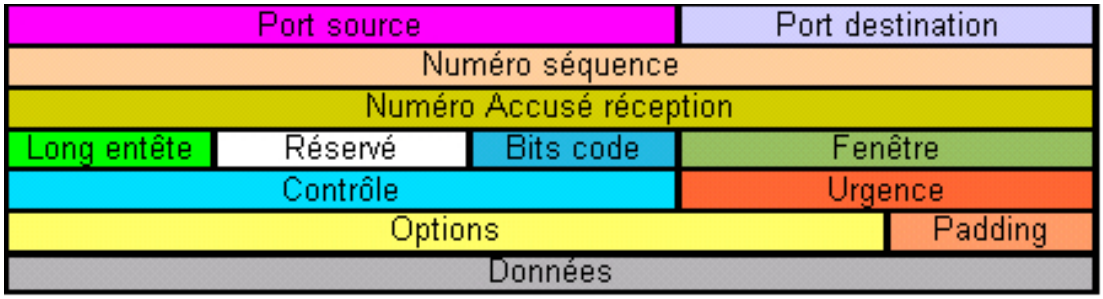

FIGURE 7.4 – Structure d'un en-tête TCP

Le schéma ci dessus représente la partie données de l'entête IP vue précédemment, pour TCP.

Ports ports locaux utilisés

Numéro de séquence un flot est découpé en segments qui seront numérotés de manière incrémentale

Numéro accusé de réception indique que l'on a bien reçu le segment précédent

Long entête multiple de 32 bits, permet de savoir où commencent les données.

Réservé RAS

Bits de code 6 bits utilisés comme drapeaux indiquant des informations supplémentaires sur la nature du paquet :

URG données urgentes

ACK accusé de réception

PSH données ne doivent pas être mises en buffer

RST reset demandé

SYN demande de synchronisation des numéros de séquence en début de communication

FIN signale la fin du flot

Fenêtre entier indiquant le nombre de segments pouvant être envoyés avant d'envoyer un accusé de réception

Contrôle RAS

Urgence Si le drapeau URG est positionné, contient le nombre de segments à venir avant que les données ne soient plus urgentes

Options voir datagramme IP

Padding dito

UDP (User Datagram Protocol : transmission en mode non connecté) : Ici , l'analogie est faite avec l'envoi d'un courrier postal : il y a juste envoi d'un poste à un autre *sans aérification* que le client a effectivement reçu cette information : des pertes de données sont possibles, mais ce protocole a l'avantage d'être beaucoup plus léger et rapide que TCP, de part le faible nombre d'informations et de vérifications supplémentaires à traiter.

Avec UDP, la remise en ordre et la vérification de la validité des messages reçus est laissée aux couche supérieures, si elles souhaitent l'implémenter :

Si l'ordre des paquets est important pour un flux vidéo par exemple, la perte d'une image est rarement dramatique, car l'erreur sera pratiquement invisible à l'affichage.

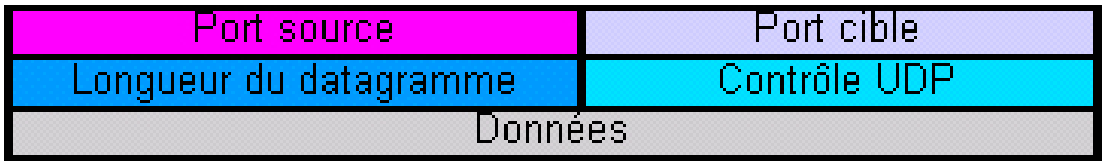

FIGURE 7.5 – Structure d'un envoi UDP

Le schéma ci dessus représente la partie données de l'entête IP vue précédemment, pour UDP.

Comme on le constate ci -dessus, le protocole UDP est plus simple et ne nécessite pas d'explications complémentaires.

Les sockets Lorsqu'un ordinateur envoie un message à un autre ordinateur, il s'adresse à une application. En règle générale plusieurs applications cohabitent sur une même machine : Il est donc nécessaire de préciser un couple "*machine-application*" auquel adresser les données, qui se traduit avec le protocole IP par le couple "adresse,port".

Le port est un numéro entier positif unique sur la machine que l'application va utiliser afin de ne récupérer que les messages la concernant : On peut comparer ce système à un immeuble de 20 appartements, l'adresse est la même mais chacun a sa boîte aux lettres (il est possible pour une même application d'occuper plusieurs ports)

ce couple (adresse IP - port) se nomme socket

Les sockets sont "situées" entre le niveau communication et le niveau application

TOUS LES PROTOCOLES CI-DESSUS SONT NECESSAIRES A L'UTILISATION DE TCP/IP ET SONT DONC PRESENTS DANS L'ENSEMBLE DES IMPLEMENTENTATIONS

Le niveau Application

C'est ici que se trouvent les protocoles le plus familiers, visibles à l'utilisateur à travers des applications dont le nom est parfois identique au nom du protocole.

Certains protocoles, si aucun port n'est explicitement précisé par l'utilisateur, tenteront de communiquer sur un port par défaut, défini par convention. Dans ce cas, celui-ci est noté ci dessous entre parenthèses : Tout autre port peut cependant être utilisé.

Nous avons déjà parlé dans la présentation de FTP (21 ou 20). POP3 (110) et SMTP (25) sont également régulièrement utilisés car à la base de l'échange de courriers électroniques.

Voici d'autres protocoles que vous utilisez ou dont vous avez peut-être déjà entendu parlé :

TFPT Trivial File Transfert Protocol (transfert de fichier "trivial" : manipulation de base sur les fichiers = envoi et réception)

SNMP Simple Network Management Protocol (protocle de gestion d'équipements réseau)

HTTP (80) Hyper Text Transfert Protocol ... c'est celui qui permet d'accéder à Moodle !

IMAP4 (173) Internet Message Access Protocol : comme POP3 permet de récupérer son courrier sur un serveur mais permet en plus des interactions entre le serveur de messagerie et le client (par exemple, conservation du courrier sur le serveur)

LDAP Light Directory Access Protocol : accès à un serveur d'annuaire normalisé X500.

NNTP (119) News Network Transport Protocol : protocole d'accès aux newsgroups (groupes de discussion) et aux forums

IRC Internet Relay Chat : c'est le protocole qui permet une communication synchrone uniquement textuelle (chat)

Tous ces protocoles sont basés sur TCP/IP.

7.2.4 Les services TCP/IP

TCP/IP est livré avec des services de base qui sont en fait des applications qui se basant sur les protocoles des couches IP et TCP.

Les applications de base (dont certaines ont été vues en commandes UNIX)

finger affiche les informations sur les personnes connectées

telnet permet d'ouvrir un terminal à distance sur une machine

ftp transfert de fichier (présenté au niveau protocole page précédente.)

mail commande support de SMTP

A côté de ces applications conçues pour TCP/IP, existent des applications dont le nom comme par "**r**" (r comme **remote**, ce qui signifie "distant" en anglais).

Ces applications permettent d'accéder à d'autres machines par des commandes analogues à celles utilisées en local. On trouve ainsi rlogin (remote login : login à distance), rsh (remote shell) , rcp (remote copy) ...

Ces outils fonctionnent sur le principe du client-serveur : Pour pouvoir utiliser la commande "ftp machinex", il est ainsi nécessaire qu'un serveur ftp tourne sur la machine `machinex`.

Sur le serveur, le processus en attente se nomme un *demon*. Par convention, cette fonction est souvent indiquée par un **"d"** à la fin du nom de service : Ainsi ftp**d** indique un démon ftp, telnet**d** un démon telnet et ainsi de suite.

Les outils de TCP/IP

⇒ `hostname` : donne le nom de la machine

⇒ `arp` : gère et affiche les informations de décodage des adresses IP en adresses machine

⇒ `netstat` : affiche l'état des connexions en cours

⇒ `ping` : permet de s'assurer que la machine que l'on souhaite atteindre répond à des requêtes réseau.

⇒ `tracert ou traceroute` : indique le chemin emprunté pour aller de votre machine à celle désignée

⇒ `ifconfig` : permet à la fois de visualiser les paramètres de configuration IP et de les positionner

⇒ `route` : permet de définir des "routes" réseau.

7.3 L'adressage

7.3.1 L'adresse IP

A l'intérieur d'un réseau tout adresse IP doit être *UNIQUE* (elle ne peut en aucun cas être attribuée à plusieurs machines d'un même réseau). Chaque machine reliée à ce réseau peut disposer d'une ou de plusieurs adresses IP, suivant le nombre de cartes réseaux dont elle dispose et au nombre.

Cette adresse sera utilisée par les protocoles TCP/IP pour communiquer avec cette machine. Actuellement, la version IP utilisée est la version 4 *(IP V4)* :

Nous ne ferons qu'évoquer la version 6, sachant que cette version est encore loin d'être utilisée. Les principes de codification de l'adresse sont donc ceux de la version 4.

Les adresses IPv4 sont codées sur 4 octets

Elles sont souvent notées en décimal, la valeur de chaque octet étant séparée par un point de la manière suivante : w.x.y.z.

Par exemple :

```
192.25.168.115
```

Chaque adresse est découpée en deux partie, une adresse fixe identifiant le réseau et une partie variable permet d'adresser les machines de celui-ci. Afin de permettre la reconnaissance de l'identifiant réseau dans l'adresse, des classes d'adresses ont été définies.

Il existe 5 classes, notées de A à E, dont les codifications sont les suivantes :

classe A identifiant réseau codé sur le 1er octet ; ses valeurs vont de 1 à 126 et l'adresse IP aura la structure suivante : [1-126].x.y.z où x.y.z est l'adresse de la machine dans le réseau plus de 16 millions de machines par réseau [(256*256*256) -2 = 16 777 214] , 126 réseaux

classe B identifiant réseau codé sur 2 octets ; ses valeurs vont de 128.0 à 191.255 et l'adresse IP aura la structure : [128-191].[0-255].y.z où y.z est l'adresse de la machine plus de 65 000 machines par réseau [(256*256) -2 = 65 534), plus de 16 000 réseaux (16 383)

classe C identifiant réseau codé sur 3 octets ; ses valeurs vont de 192.0.0 à 223.255.255 ; la structutre de l'adresse IP est : [192-223].[0-255].[0-255].z

254 machines par réseau, plus de 2 000 000 réseaux [32*256*256 = 2 097 152]

et aussi les classes :

classe D utilisée pour les diffusions en multicast. Le premier octet a une valeur comprise entre 224 et 239. La structutre de l'adresse est la suivante :

[224-239].x.y.z x.y.z représente cette fois l'adresse du groupe à atteindre (cette classe d'adresse est utilisée pour les applications de type visioconférence par exemple).

classe E le premier octet a une valeur comprise entre 240 et 255 : cette plage est réservée à l'IETF (Internet Engineering Task Force qui est l'organisme qui travaille à la définition des normes IP)

Le mode de fonctionnement des deux dernières classes étant particulier, nous ne verrons que celui concernant les classes A, B ou C.

Il est possible de savoir à quelle classe une adresse appartient en fonction de son appartenance à l'un de ces intervalles : ainsi l'adresse 192.168.99.15 est une adresse appartenant à un réseau de classe C.

Attention, la classe 127 n'est jamais émise sur le réseau. Elle est réservée aux communications inter-processus sur la machine locale. L'adresse de boucle locale est 127.0.0.1 (adresse de loopback)

Attention, la classe 0.0.0.0 signifie "ce réseau" : c'est le réseau où on se trouve

Cette dernière identification est utilisée par exemple pour les machines sans disque au moment du boot

Pour chaque adresse de la classe C, z est compris entre 0 à 255,
Pour la classe B, y.z est compris entre 0.0 et 255.255,
Pour la classe A, x.y.z, est compris entre 0.0.0 et 255.255.255.

La valeur **0** (Classe A), **0.0** (classe B), **0.0.0** (classe C) représente le numéro du réseau.

La valeur **255.255.255** (Classe A), **255.255** (classe B), **255** (classe C) signifie "*toutes les machines du réseau*" : c'est l'adresse de broadcast. Pour une machine donnée, une des valeurs entre ces bornes sera utilisée.

L'utilisation d'une adresse IP pour un réseau local ne communiquant pas avec l'extérieur est libre. Dans ce cas, on utilise des adresses dites "de réseau privé" qui correspondent pour chaque classe A, B ou C à des adresses qui ne seront jamais routées d'un réseau à l'autre.

Il s'agit des adresses commençant par

```
10.0.0.0 à 10.255.255.255 pour la classe A
172.16.0.0 à 172.31.255.255 pour la classe B
192.168.0.0 à 192.168.255.255 pour la classe C
```

Par contre, tout réseau devant communiquer avec l'extérieur doit avoir une adresse attribuée par le NIC (Network Information Center). IPv4 permettant de coder un total de 4 milliards d'adresse, une pénurie d'adresse commence à apparaître depuis plusieurs années suite à l'augmentation du nombre d'équipements reliés à Internet : Afin de pallier ce problème, la version IPv6 a adopté un principe d'adresse codées sur 16 octets avec une catégorisation de classes différentes.

Une adresse IP V6 est de type

```
AD3E:F0D1:25F1:53BC:551F:3FE4:DA41:11DD
```

On voit imédiatement que la transition d'IPv4 à IPv6 n'est pas sans conséquences, du fait du changement de l'incompatibilité des deux types d'adresses : Nous allons voir comment, en parallèle à l'arrivée de la version 6, l'attribution des plages d'adresses a permis à IPv4 de s'adapter à la demande croissante tout en permettant de l'ancien adressage.

7.3.2 Les masques

Les masques de sous réseaux (subnet mask)

Il est fréquent de devoir créer des sous réseaux à l'intérieur d'une entreprise. Imaginons qu'une adresse de classe A "12.x.y.z" a été attribuée. Afin de créer plusieurs sous réseaux , seront utilisées des "pseudo" classes inférieures assorties d'un masque de "sous réseau". Ainsi, ce n'est plus l'adresse 12.x.y.z qui sera utilisée, mais l'adresse 12.15.y.z qui est une pseudo adresse B.

Principe général du masque

Les adresses IP étant composées d'une partie identifiant de réseau et d'une partie identifiant de machine, il est nécessaire d'appliquer un masque sur ces adresses afin de déterminer la portion qui identifie une machine dans le réseau. L'application à toute adresse IP d'un masque en effectuant une opération correspondant à un **ET logique** sur chaque bit permet de trouver l'ID du réseau.

Par défaut, chaque classe d'adresse possède un masque prédéfini :
$\Rightarrow$ pour la classe A 255.0.0.0
$\Rightarrow$ pour la classe B 255.255.0.0
$\Rightarrow$ pour la classe C 255.255.255.0
exempl : adresse 180.10.12.1 cette adresse est une adresse de classe B. Le masque par défaut à appliquer est donc 255.255.0.0. Voici l'opération ET à effectuer :

Le numéro du réseau *180.10.0.0*, l'adresse de la machine dans ce réseau est*12.1*

TABLE 7.2 – Application du masque réseau

	Notation décimale	Notation binaire
Adresse IP	180.10.12.1	10110100 00001010 00001100 00000001
Masque	255.255.0.0	11111111 11111111 00000000 00000000
Après le ET logique	180.10.0.0	10110100 00001010 00000000 00000000

Toutes machines pour lesquelle l'application du masque donnera le réseau 180.10.0.0 seront donc dans le même réseau : Elles pourront communiquer entre elles directement.

Lorsque la machine émettrice appartient au réseau 180.10.0.0 , c'est ARP qui prend le relais pour savoir quelle est la machine qui a l'adresse 12.1 et transformer cette adresse IP en adresse Mac. Pour ce faire, ARP envoie sur le réseau local une requête par broadcast, c'est à dire à l'adresse 180.10.255.255 : toutes les machines reçoivent le message et celle dont l'adresse correspond répond. Si la machine émettrice n'est pas sur le même réseau, elle s'adresse au routeur afin de determiner quel chemin prendre pour atteindre la machine 180.10.12.1 .

Le routeur est aussi appelé *default gateway* (passerelle par défaut). Son adresse IP doit être sur le même réseau que les machines qui s'adresseront à lui et être précisée lors de la configuration de chacune des machines (lors de l'installation ou ultérieurement).

Les trois éléments qui permettent de communiquer avec une machine sont
⇒ le masque de sous réseau qui s'applique
⇒ l'adresse du routeur auquel il faut s'adresser pour changer de réseau
⇒ son adresse
exemple :

Machine A
IP : 192.168.4.3

A envoie un message à B :
- A possède une adresse de classe C
  192.168.4.3 ET 255.255.255.0
  => A est dans le réseau 192.168.4.0
- B possède une adresse de classe C
  192.168.1.5 ET 255.255.255.0
  => B est dans le réseau 192.168.1.0

A et B ne sont pas dans le même réseau : A envoie à sa passerelle.

Machine B
IP : 192.168.1.5

FIGURE 7.6 – Communication entre 2 ordinateurs

Caractéristiques d'un routeur

Le routeur (ou gateway ou passerelle) permet de passer d'un réseau à un autre.

TABLE 7.3 – Configuration d'un routeur

Port 1	Port 2
Adresse Mac 1	Adresse mac 2
Adresse IP sur réseau A	Adresse IP sur autre réseau

Dans l'exemple ci dessus, l'adresse IP du routeur sur le réseau A pourrait être 26.0.0.12.

Pour atteindre la machine située sur le réseau 28.0.0.0 il doit s'adresser à un autre routeur car il faut changer de réseau IP. Pour cefaire, le routeur possède une "table de routage" dans laquelle sont répertoriées les adresses IP des routeurs auquels il pourra s'adresser en fonction de la machine à atteindre.

Le subnetting ou l'utilisation d'un masque de sous réseau

Nous avons mentionné la possibilité de découper un réseau en sous-réseaux. Cette approche est souvent utilisée en entreprise pour les raisons suivantes :

- ⇒ éviter de polluer les réseaux avec les requêtes de type broadcast : plus les sous réseaux auront un nombre réduit de machines, moins les messages broadcast auront d'influence sur le débit.
- ⇒ permettre de changer de réseau physique (cas de réseaux locaux interconnectés par des réseaux "voie publique").
- ⇒ isoler les différents secteurs d'une entreprise, pour des raisons de sécurité.

exemple : Rappel : Le masque par défaut de la classe C est 255.255.255.0 et ne permet d'adresser qu'un maximum que 254 machines (rappel : adresse 0 = le réseau, adresse 255 = le broadcast). Un masque de sous réseau commence comme le masque réseau mais utilise dans le cas de la classe C le dernier octet. Afin de multiplier les possibilités d'adressage, on peut convenir de découper une adresse de classe C `199.110.31.y` en 4 plages.

On peut ainsi mettre en place un premier masque de sous réseau pour le premier niveau de découpage qui serait 255.255.255.192 (11111111.11111111.11111111.11**000000**).

Le champ réservé pour l'adresse de la machine fait donc 6 bits. L'adresse à zéro correspond à l'adresse réseau, ce qui signifie que toutes les valeurs de l'octet 4 qui correspondent à ces 6 bits à zéro seront des adresses de réseau comme :

```
199.110.31.64  11000111 01101110 00011111 01000000
199.110.31.128 11000111 01101110 00011111 10000000
```

```
199.110.31.192 11000111 01101110 00011111 11000000
```

et bien sûr 199.110.31.0

Les valeurs positionnant les 6 bits à 1 sont des adresses de broadcast. Nous aurons donc avec un masque de sous réseau personnalisé de 255.255.255.192 la possibililité de créer 4 de sous réseaux.

Exemple du premier sous-réseau :

```
masque 255.255.255.192
11111111.11111111.11111111.11000000
sous réseau 199.110.31.64
11000111 01101110 00011111 01000000
broadcast 199.110.31.127
11000111 01101110 00011111 01111111

1ère machine adressable 199.110.31.65
11000111 01101110 00011111 01000001
dernière machine adressable 199.110.31.126
11000111 01101110 00011111 01111110
```

soit 62 machines pour ce sous réseau.

Nous pourrions créer des sous réseaux en utilisant par exemple le masque 255.255.255.252 (11111111.11111111.11111111.11111100) On aurait alors 64 sous réseaux (2^6) avec chacun 2 postes soit 64 X 2 = 128 machines adressables.

Le CIDR - Classless Inter Domain Routing

Confrontés à la pénurie prévisible d'adresses, les organismes chargés d'allouer les adresses ont finallement abandonné le découpage en classes que nous avons vu précédemment. Est né le "CIDR" qui a pour but :

1. d'alléger les tables de routage sur les routeurs
2. d'optimiser l'allocation d'adresses et d'éviter le gaspillage induit par les classes

Ainsi ce n'est plus une adresse, avec une classe implicite, mais un lot d'adresses accompagnées d'un masque de sous réseau associé qui couvrira ces adresses qui est alloué.

Exemple : je souhaite 200 adresses

Me sera donné l'adresse 206.65.12.0 ainsi que le masque de sous réseaux 255.255.254.0 ce qui en binaire donne :

pour l'adresse : 11001110 01000001 00001100 00000000

pour le masque 11111111 11111111 1111111**0** 00000000

On voit dans cet exemple que le masque de sous réseau donné comporte (en rouge) les **23** bits décrivant le sous réseau, les **9** suivants (en vert) serviront à numéroter les hôtes de ce sous réseau. La notation CIDR décrivant l'adresse et le sous réseau correspondant s'écrit : 206.65.12.0/**23** , le "23" faisant référence au nombre de bits à 1 du masque.

7.4 Les fichiers de configuration de TCP/IP

Nous voyons ici les différents fichiers supports de TCP/IP et leur rôle : Les fichiers de configuration se trouvent dans le répertoire /etc, sur un sytème Unix.

7.4.1 Les fichiers de configuration TCP/IP

Le fichier `/etc/hosts` permet de déclarer pour chaque adresse IP un nom de machine qui permettra ensuite de désigner cette machine par son nom. Il est en effet plus facile de se souvenir qu'une machine s'appelle par exemple "messagerie" que de son adresse IP :

Il est possible d'assigner plusieurs noms de machines à une même adresse IP (localhost et le nom donné à la machine locale, par exemple).

Contenu de `/etc/hosts` :

```
192.25.66.105 messagerie mess MESSAGERIE # serveur de messagerie
```

Le premier champ est l'adressse IP, les suivants sont les différents noms succeptibles d'être utilisés pour cette machine. Après le # se trouvent les commentaires.

Le fichier `/etc/services` permet de recenser les numéros de port utilisés par les différentes applications.

- ⇒ Les ports 1 à 1023 sont réservés aux applications les plus connues, comme ftp qui utilise le port 21 sur la plupart des machines. Il est nécessaire d'avoir les permissions root pour les utiliser.
- ⇒ Les ports 1024 à 5000 sont affectés par le système en l'absence de précisions
- ⇒ Les ports au delà de 6000 sont ouverts à toutes les autres application

Contenu de `/etc/services` :

```
hostname 101/tcp hostname
route 520/udp routed # pour savoir quel chemin est emprunté
ftp 21/tcp
```

Le premier champ est le nom du service, ensuite viennent le numéro de port utilisé et le protocole impliqué (tcp ou udp) . Il peut exister des alias pour le service (comme `routed` pour `route`) et il est conseillé de rajouter des commentaires après un # .

Le fichier `/etc/networks` permet quant à lui de donner un nom à un réseau (à la manière de `/etc/hosts`)

Contenu de `/etc/networks` :

```
loop 127 loopback # réseau pour s'auto adresser
univ-fcomte.fr 194.57 # universite de Franche Comte
```

7.4.2 Les fichiers d'autorisation

Nous avons vu qu'il existait des commandes "r" (*remote*) qui permettent de se connecter et d'exécuter des traitements sur une machine distante.

Pour des raisons évidente de sécurité d'accès, ces commandes nécessitent la mise en place d'autorisations afin qu'il ne soit pas suffisant de connaître l'adresse IP d'une machine pour pouvoir s'y connecter et lui soumettre des instructions.

> *La déclaration des machines clientes doit impérativement être faite dans le fichier /etc/hosts du serveur.*

Deux fichiers principaux sont utilisés sur la machine serveur :

⇒ `/etc/hosts.equiv` permet de définir les noms des machines pour lesquelles on autorise l'identification aux utilisateurs existants sur les deux machines (il faut que l'utilisateur soit le même sur les 2 machines). Cette possibilité n'est néanmoins pas autorisée pour les utilisateurs dont l'UID est égal à zéro (comme root)

⇒ ∼user/.rhosts [1] permet à chaque utilisateur de définir les couples utilisateur/-machine qu'il souhaite autoriser dans son environnement. Attention, le fichier .rhosts ne peut fonctionner que s'il est accessible en écriture à son propriétaire !

Un autre fichier est mis à la disposition des utilisateurs pour leur permettre cette fois d'entrer les login/password dont ils ont besoin pour accéder à des services sur des machines distantes. Il s'agit de

⇒ ∼`user/.netrc` dans lequel sont précisés : le nom de la machine, le login et le password nécessaire pour cette machine

Exemple

```
machine messagerie login georges password coucou
```

L'implémentation de ce fichier permet ensuite à `georges` par exemple de lancer la commande `ftp messagerie` et de se retrouver directement sur son répertoire d'accueil ftp de la machine messagerie sans avoir besoin de s'identifier.

1. ∼user est un raccourci permettant de désigner le repertoire d'accueil d'un utilisateur. Ainsi, pour un utilisateur georges, son répertoire d'accueil est directement accessible en faisant cd ∼georges , pour aller dans le répertoire de root, cd ∼root

Ce principe est séduisant, néanmoins il présente le désavantage de permettre le stockage en clair de son mot de passe dans un fichier, bien que le fichier .netrc soit en `-rw-------`

Ces fichiers sont les mécanismes de base UNIX. D'autres produits ont été développés et sont utilisés pour permettre la gestion de login sur des machines différentes en assurant la sécurité de l'ensemble. L'outil le plus connu et disponible sous Linux est NIS de la société SunMicrosystems. Il a un successeur de plus en plus utilisé NIS+.

La sécurité des sytèmes est un très vaste sujet, nécessitant à elle seule un cours conséquent. Outre NIS cité plus haut qui permet de répertorier les utilisateurs et les machines d'un parc ainsi que les droits afférents, on pourra citer KERBEROS qui fonctionne selon un principe de "tickets d'accès" limités dans le temps qui est une solution répandue.

7.5 Exercices

Exercice 17 : Classes d'adresses et masques réseaux

Déterminer d'après le masque quel est le réseau utilisé en utilisant la notation binaire pour les trois adresses suivantes.

⇒ Adresse 125.24.6.2
⇒ Adresse 130.14.28.2
⇒ Adresse 125.24.6.2 cette fois avec le masque 255.255.254.0

Exercice 18 : Communications machine à machine

Considérons ces deux adresses IP : 138.12.3.8 et 138.12.7.25

Question 18.1 : Cas A : sans autre indication ces deux machines sont-elles sur le même réseau ?

Question 18.2 : Cas B : le masque précisé est 255.255.255.0 : ces deux machines sont-elles sur le même réseau ?

Question 18.3 : A quelle condition ces deux machines peuvent-elles communiquer dans le cas A et dans le cas B ?

Question 18.4 : Quelle règle simple peut-on déduire pour connaître le réseau sans passer par le binaire pour un masque par défaut ?

Exercice 19 : Utilisation des commandes ping et traceroute

TCP/IP existant sur UNIX et sur Windows, sur ce dernier, en ouvrant une invite de commande (démarrer , exécuter , "command" ou "cmd" ; icône "commande MS-DOS ...) vous avez accès aux commandes `tracert` et `ping`.

Vous allez exécuter ces commandes lorsque vous êtes connectés sur l'internet en donnant en paramètre l'adresse du CTU soit

```
ping cvfc.univ-fcomte.fr
traceroute cvfc.univ-fcomte.fr
```

Les réponses dans le cas de la seconde commande différeront d'une personne à l'autre. Observez les réponses ... (pour la commande ping, ceux qui sont sous UNIX doivent interrompre l'affichage par CTRL-C ; sous windows, par défaut seules 4 lignes sont affichées)

Exercice 20 : Découpage de plages réseaux en sous-réseaux

Dans le cadre de la mise en place d'un réseaux comprenant quatre salles de travaux pratiques, on souhaite allouer un sous-réseau pièce, connectés au moyen de deux routeurs gérant chacun deux sous-réseaux de machines.

Voici les nombres de machines à prévoir pour chaque salle de TP ainsi que le nom du routeur auquel celle-ci devra être raccordée :

- ⇒ 35 postes dans la salle 100A (routeur A)
- ⇒ 24 postes dans la salle 101A (routeur A)
- ⇒ 43 postes dans la salle 200B (routeur B)
- ⇒ 12 postes dans la salle 205B (routeur B)

Pour ce découpage, on propose de prendre le réseau IP (en notation CIDR) suivant comme base de découpage : 192.168.3.0/24.

Question 20.1 : Quel est le nombre total d'adresses qui seront utilisées pour mettre en place ce réseau ?

Question 20.2 : Combien de sous-réseaux seront-ils nécessaires, au total, en comptant l'interconnexion entre les salles ? Quelle taille retenir pour chacun de ces sous-réseau ?

Question 20.3 : Donner la liste des sous-réseaux obtenus et de leurs informations (adresse du réseau, masque en notation décimale et CIDR, adresse de la passerelle s'il y a lieu, adresse de diffusion).

Question 20.4 : Établir un schéma de l'architecture réseau obtenue.

7.6 Correction des exercices

Correction Exercice 17 : Classes d'adresses et masques réseaux

L' adresse 125.24.6.2 est une adresse de classe A (125 < 128) - Le masque qui s'applique est le 255.0.0.0

```
valeur de l'adresse : 01111101.00011000.00000110.00000010
masque : 11111111.00000000.00000000.00000000
réseau : 01111101.00000000.00000000.00000000
Réseau = 125.0.0.0
```

l' adresse 130.14.28.2 est une adresse de classe B (127 < 130 < 192)- Le masque qui s'applique est le 255.255.0.0

```
valeur de l'adresse : 10000010.00001110.00011100.00000010
masque : 11111111.11111111.00000000.00000000
réseau : 10000010.00001110.00000000.00000000
Réseau = 130.14.0.0

l' adresse 125.24.6.2 est un adresse de classe A -
Cependant, un masque de sous réseau est précisé
255.255.254.
valeur de l'adresse : 01111101.00011000.00000110.00000010
masque : 11111111.11111111.11111110.00000000
réseau : 01111101.00011000.00000110.00000000
Réseau = 125.24.6.0
```

Correction Exercice 18 : Communications machine à machine

Solution question 18.1 :

Aucun masque n'est précisé. L'adresse commençant par "138" qui est compris entre 128 et 191, il s'agit d'une adresse de classe B. C'est le masque 255.255.0.0 qui s'applique par défaut.

```
Adresse 138.12.3.8
valeur de l'adresse : 10001010.00001100.00000011.00001000
masque : 11111111.11111111.00000000.00000000
réseau : 10001010.00001100.00000000.00000000
```

Il s'agit de la machine 3.8 sur le réseau 138.12

```
Adresse 138.12.7.25
valeur de l'adresse : 10001010.00001100.00000111.00011001
masque : 11111111.11111111.00000000.00000000
réseau : 10001010.00001100.00000000.00000000
```

Il s'agit de la machine 7.25 sur le réseau 138.12

Solution question 18.2 :

Le masque 255.255.255.0 est précisé . Lorsqu'on l'applique on obtient :

```
Adresse 138.12.3.8
valeur de l'adresse : 10001010.00001100.00000011.00001000
masque : 11111111.11111111.11111111.00000000
réseau : 10001010.00001100.00000011.00000000
```

Il s'agit de la machine 8 sur le réseau 138.12.3

```
Adresse 138.12.7.25
valeur de l'adresse : 10001010.00001100.00000111.00011001
masque : 11111111.11111111.11111111.00000000
réseau : 10001010.00001100.00000111.00000000
```

Il s'agit de la machine 25 sur le réseau 138.12.7

Solution question 18.3 :

Dans le cas A , les deux machines étant sur le même réseau, elles communiquent entre elles directement. Dans le cas B, les deux machines n'étant pas sur le même réseau, il faut un routeur pour qu'elles puissent communiquer.

Les masques par défaut comportent des parties où les bits sont tous à "1" (255) et des parties où les bits sont tous à zéro (0).

Lorsque les bits sont tous à "1" , on constate que la valeur de l'octet est intégralement reconduite, dans les cas contraire, la valeur de l'octet est à zéro.

Solution question 18.4 :

On peut donc simplement en déduire que les bits à 1 "laissent passer" les valeurs, les bits à 0 "bloquent" .

Ainsi, sur une adresse où w.x.y.z = 138.61.25.124 , le masque par défaut étant 255.255.0.0 w et x "passent" et représentent le réseau, y et z "bloquent" et représentent la machine

```
adresse 138. 61.25.124
masque  255.255. 0.  0
réseau  138. 61
machine 25 .124
```

Correction Exercice 19 : Utilisation des commandes ping et traceroute

Cet exercice n'a pas de correction puisqu'il s'agit d'utiliser des commandes TCP/IP sur votre machine.

Correction Exercice 20 : Découpage de plages réseaux en sous-réseaux

Solution question 20.1 :

Pour chaque sous-réseau, il est nécessaire de prévoir à la fois le nombre de postes indiqués mais également une adresse supplémentaire attribuée à l'interface du routeur responsable de l'interconnexion :

$35 + 24 + 43 + 12 + 4 \times 1 = 118$ adresses.

A ces 118 adresses il ne faut pas oublier d'ajouter deux adresses supplémentaires pour permettre de gérer l'interconnexion entre les deux routeurs (voir schéma ci-après) :

Nous aurons donc au total $118 + 2 = 120$ adresses utilisées.

Solution question 20.2 :

Au total, 5 sous-réseaux sont nécessaires : 4 pour les salles de travaux pratiques (un par salle) + 1 sous-réseau d'interconnexion entre les deux routeurs demandés dans l'énoncé.

Par commodité, nous ferons référence à ces sous-réseaux dans la suite des réponses en utilisant les noms SR1, SR2, SR3, SR4 et SR5.

Voici les tailles à sélectionner pour chacun de ces sous-réseaux :

- $\Rightarrow$ SR1 (salle de TP 100A) : 35 postes + 1 adresse routeur + 2 adresses réservées = 38 adresses requises. Pour déterminer la taille du sous-réseau, on retient la puissance de deux immédiatement supérieure à ce chiffre $64\ (2^6)$, de manière à gâcher un minimum d'adresses (découpage au plus juste).
- $\Rightarrow$ SR2 (salle de TP 101A) : 24 postes + 1 adresse routeur + 2 adresses réservées = 27 adresses requises. En appliquant le même raisonnement que pour SR1, cela nous donne une taille de $32\ (2^5)$ adresses.

⇒ SR3 (salle de TP 200B) : 43 postes + 1 adresse routeur + 2 adresses réservées = 44 adresses requises. Taille de sous-réseau retenue : 64 (2^6) adresses.
⇒ SR4 (salle de TP 205B) : 12 postes + 1 adresse routeur + 2 adresses réservées = 15 adresses requises. Taille de sous-réseau retenue : 16 (2^4) adresses.
⇒ SR5 (interconnexion routeurs) : 2 routeurs + 2 adresses réservées = 4 adresses requises. Taille de sous-réseau retenue : 4 (2^2) adresses.

Solution question 20.3 :

Pour éviter des éventuels problèmes d'alignement réseau, on choisit de découper nos sous-réseaux par ordre de taille décroissante.

SR1

```
Adresse réseau : 192.168.3.0/26 (26 = 32 - 6)
Masque réseau en décimal : 255.255.255.192
Adresse de la passerelle : 192.168.3.62
Adresse de diffusion : 192.168.3.63
```

SR3

```
Adresse réseau : 192.168.3.64/26 (26 = 32 - 6)
Masque réseau en décimal : 255.255.255.192
Adresse de la passerelle : 192.168.3.126
Adresse de diffusion : 192.168.3.127
```

SR2

```
Adresse réseau : 192.168.3.128/27 (27 = 32 - 5)
Masque réseau en décimal : 255.255.255.224
Adresse de la passerelle : 192.168.3.158
Adresse de diffusion : 192.168.3.159
```

SR4

```
Adresse réseau : 192.168.3.160/28 (28 = 32 - 4)
Masque réseau en décimal : 255.255.255.240
Adresse de la passerelle : 192.168.3.174
Adresse de diffusion : 192.168.3.3.175
```

SR5

```
Adresse réseau : 192.168.3.176/30 (30 = 32 - 2)
Masque réseau en décimal : 255.255.255.252
Adresse de la paserelle : Non applicable (réseau de routeurs)
Adresse de diffusion : 192.168.3.179
```

Solution question 20.4 :

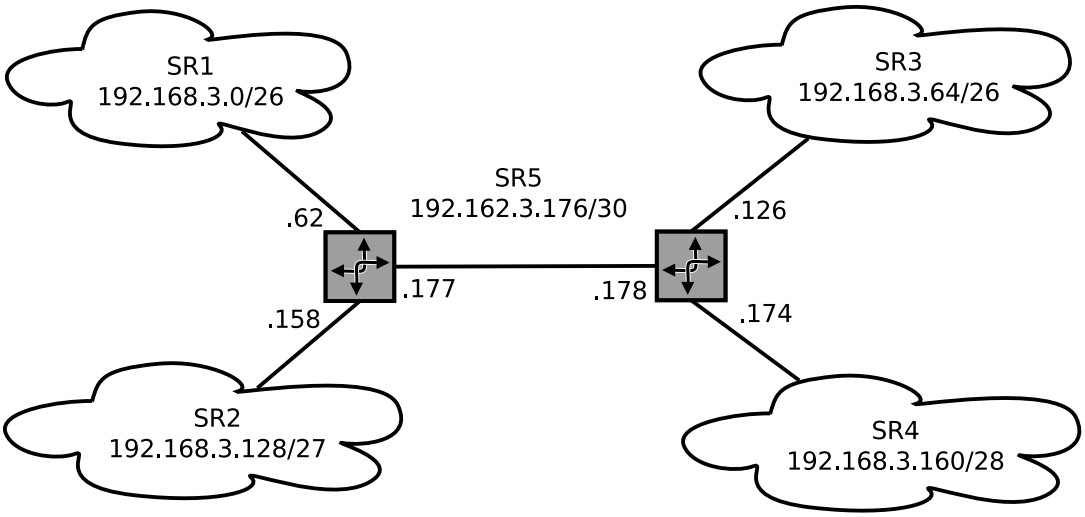

FIGURE 7.7 – Schéma du découpage réseau obtenu

Chapitre 8

L'administration et la programmation système - BH -

Ce chapitre est composé de deux parties : administration système et programmation système. Dans la première partie nous présentons comment gérer l'espace disque et comment le système Unix démarre. Dans la seconde partie, nous introduisons la programmation système en langage C, en prenant pour exemple le passage de paramètres d'un programme, la création de processus et l'accès aux fichiers.

8.1 Administration système

8.1.1 La gestion de l'espace disque

L'exécution de processus peut être à l'origine de la création de divers fichiers qui, s'ils ne sont jamais épurés, peuvent finir par prendre beaucoup de place, voire bloquer le système (nous avons vu que les diverses limites permettent d'éviter d'en arriver là !)

Parmi les fichiers à épurer, il y a les résultats d'exécution, tous les fichiers que vous aurez créés par redirection des sorties standard. Lorsqu'un programme s'aborte, l'image du programme en mémoire se stocke sur votre disque dur dans un fichier nommé `core`, ce fichier peut être volumineux, et l'exploration de cette image est rarement faite.

Les deux commandes suivantes permettent de visualiser l'utilisation de l'espace disque.

La commande `du` :

```
du [option] [repertoire]
```

Cette commande affiche la quantité d'espace disque occupée par chacun de ses arguments. La taille des fichiers et des répertoires est exprimée en blocks (dans le cas

présent, les blocks sont de 1 Ko - Ce cours n'aborde pas cet aspect des choses. Retenez simplement que sous les Linux® vous avez des blocks de 1 Ko).

Sans option, la commande `du` affiche la taille des **répertoires** à partir de l'endroit où vous êtes dans l'arborescence (`du` est équivalent à `du .`). Vous pouvez indiquer le nom d'un répertoire à analyser.

Listing 8.1– Exemple d'utilisation de la commande du

```
$ du
8 ./Desktop/Autostart
28 ./Desktop/Templates
4 ./Desktop/Trash
68 ./Desktop
4 ./.kde/share/applnk
4 ./.kde/share/apps/kdehelp
4 ./.kde/share/apps/kdisknav
4 ./.kde/share/apps/kfm/bookmarks
4 ./.kde/share/apps/kfm/tmp
12 ./.kde/share/apps/kfm
4 ./.kde/share/apps/kpanel/applnk
8 ./.kde/share/apps/kpanel
32 ./.kde/share/apps
28 ./.kde/share/config
4 ./.kde/share/icons/mini
8 ./.kde/share/icons
4 ./.kde/share/mimelnk
80 ./.kde/share
84 ./.kde
192 ./public_html
4 ./scripts
40 ./shells
52972 .
```

Sans option, la commande du affiche la taille des répertoires et sous répertoires mais pas le détail des fichiers dans ceux ci.

L'option `-a` permet d'afficher la taille des fichiers contenus dans les répertoires :

```
du -a [repertoire]
```

Listing 8.2– Exemple d'utilisation de la commande `du -a`

```
$ du -a
4 ./.bash_logout
4 ./.bash_profile
4 ./.bashrc
4 ./Desktop/Autostart/Autorun.kdelnk
8 ./Desktop/Autostart
4 ./Desktop/Printer.kdelnk
# ... lignes omises ...
44 ./telnet.txt
8 ./shells/3.jpeg
8 ./shells/1.jpeg
12 ./shells/2.jpeg
4 ./shells/index.htm
4 ./shells/count.dat
40 ./shells
26228 ./form2.ps
26228 ./form.ps
52972 .
```

*Dans cet exemple, vous avez tous les **répertoires**, et tous les **fichiers** avec leur chemin d'accès et leur taille.*

L'option `-s` réalise "l'inverse" de la précédente, c'est à dire qu'elle affiche seulement la taille du répertoire en cours ou demandé, sans aucun détail.

```
du -s [repertoire]
```

Listing 8.3– Exemple d'utilisation de la commande `du -s`

```
$ du -s
52972 .
```

La commande `df`

```
df [option] [repertoire]
```

La commande `df` indique les quantités d'espaces disque utilisés et disponibles sur les systèmes de fichiers. Sans argument, `df` indique les quantités correspondant à tous les systèmes de fichiers montés, quelques soient leurs types. Sinon, `df` affiche les données corespondant aux systèmes contenant chaque fichier fournit en argument. Cette commande est à utiliser de façon régulière pour éviter les problèmes de disque plein.

Les informations affichées par la commande df sont :

⇒ **File system** : le nom du système de fichiers : c'est un `/dev`, un "device" , c'est à dire un disque physique.
⇒ **1k-blocks** : le nombre total de blocks de 1ko disponibles (= la taille du disque en ko)
⇒ **Used :** le nombre de blocks utilisés (= la taille en ko utilisée)
⇒ **Available** : le nombre de blocks libres (= la taille en ko disponible)
⇒ **Use**`%` : pourcentage de place utilisée (évite de faire le calcul ...)
⇒ **Monted on :** Nom du point de montage pour accéder à ce file system
Exemple :

Filesystem	1k-blocks	Used	Available	Use `%`	Mounted on
/dev/sda1	1035660	631068	351984	65`%`	/
/dev/sda6	2071384	1156380	809780	59`%`	/usr
/dev/sda7	1548096	542248	927132	37`%`	/usr/local
/dev/sda8	3565664	99560	3284976	3`%`	/var/ftp
/dev/md0	6048260	2095024	3646000	37`%`	/home
/dev/md1	2846088	233912	2467600	9`%`	/var/spool/mail

8.1.2 Démarrage d'un système Unix

Nous présentons dans cette section, l'administation du démarrage d'un système Unix. Comme pour nous l'avons déjà précisé à diverses reprises, le démarrage ne sera pas exactement le même selon les versions d'Unix et les distribution de Linux.

Au démarrage de la machine, est lu le MBR (Master Boot Record) dans lequel sont stockées les informations nécessaires pour identifier le premier programme à exécuter. Il s'agit du code d'amorçage ou "loader". Le "loader" charge le noyau du système, prépare les zones mémoires nécessaires et lance le système.

Comme nous l'avons vu précédemment, le premier processus exécuté par le système est `/etc/init` qui a comme PID `1`.

Ce processus va piloter le démarrage du système Unix et pour cela activer les services du système Unix selon le niveau de démarrage souhaité passé en argument. Le système Unix propose plusieurs niveaux de démarrage différents, selon que l'administrateur souhaite activer le mode mono-utilisateur (single user) ou multi-utilisateur, activer le montage des disque réseaux ou non, etc.

Les niveaux de démarrage Les différents niveaux de démarrage du système sont présentés dans le tableau 8.1.

Il ne peut y avoir qu'un niveau actif à un moment donné.

Niveau	Signification
Niveau 0	arrêt définitif du système
Niveau 1	mono utilisateur - Le seul montage est /
Niveau 2	multi utilisateurs
Niveau 3	possibilité de monter des file system physiquement situés sur d'autres machines à travers le réseau
Niveau 4	Niveau utilisateur supplémentaire
Niveau 5	Interface graphique
Niveau 6	Arrêt redémarrage sur le noyau actuel

TABLE 8.1 – Les niveaux de démarrage du système

Le fichier /etc/inittab Le processus `init` est lancé avec un niveau donné. Pour connaître les services à exécuter à ce niveau il consulte le fichier `/etc/inittab`. En effet, dans ce fichier sont précisées les commandes à exécuter au démarrage du système selon le niveau demandé. Chaque ligne du fichier `/etc/inittab` a la structure suivante :

```
ident:niveau:contexte:commande shell
```

Le tableau 8.2 donne la signification de chacun des champs composant une ligne du fichier `/etc/inittab`.

Nom du champs	Signification
ident	identificateur de la ligne
niveau	niveau (ou les niveaux) où la commande est acceptée
contexte	contexte dans lequel la commande doit s'exécuter, ces contextes sont présentés dans le tableau 8.3
commande shell	commande à exécuter, généralement un script shell activant différents services du système liés au niveau sélectionné

TABLE 8.2 – Structure du fichier `/etc/inittab`

Le tableau 8.3 donne les différents contextes dans lesquels les commandes décrites dans le fichier `/etc/inittab` peuvent s'exécuter.

Remarque : A chaque niveau de fonctionnement ne s'excécutent que les processus de ce niveau.

Contexte	Signification
sysinit	le processus est à excécuter avant tout autre processus
bootwait	s'éxécute la première fois où le système passe du niveau 1 au niveau 2
wait	s'exécute à chaque fois qu'il y a changement de niveau
once	idem à wait mais une seule fois
initdefault	niveau par défaut au démarrage d'Unix
respawn	relance le processus à chaque fois qu'il se termine

TABLE 8.3 – Les différents contextes du fichier `inittab`

Listing 8.4– Exemple de fichier `/etc/inittab` :

```
l0:0:wait:/etc/init.d/rc 0
l1:1:wait:/etc/init.d/rc 1
l2:2:wait:/etc/init.d/rc 2
l3:3:wait:/etc/init.d/rc 3
l4:4:wait:/etc/init.d/rc 4
l5:5:wait:/etc/init.d/rc 5
l6:6:wait:/etc/init.d/rc 6
ca:12345:ctrlaltdel:/sbin/shutdown -t1 -a -r now
pf::powerwait:/etc/init.d/powerfail start
...
```

A tout moment, le super utilisateur peut exécuter la commande suivante pour changer de niveau de démarrage. :

```
/sbin/init [niveau]
```

Les fichiers /etc/rc* La majorité des commandes indiquées dans le fichier `/etc/inittab` sont des shell scripts rangés dans le répertoire `/etc`.

Ces fichiers de commandes shell sont nommés **rcx** où **x** correspond au niveau du système. Par exemple, le fichier de commandes `rc1` correspond au niveau `1`. Chaque fichier de commandes est écrit de façon à accepter les options `start` et `stop` qui permettent d'automatiser le lancement et l'arrêt de ces commandes.

Les fichiers de commandes **rcx** exécutent des actions d'initialisation puis lancent les commandes situées dans le répertoire **/etc/rcx.d** (un répertoire par niveau `x`). Par exemple, les commandes exécutées au niveau 1 sont stockées dans le répertoire ***/etc/rc1.d***.

Les noms des fichiers stockés dans les répertoires **/etc/rcx.d** sont de la forme :
⇒ S`nn`*nom* : S comme "Start" : ce ficher est exécuté avec l'option `start`
⇒ K`nn`*nom* : K comme "Kill" : ce ficher est exécuté avec l'option `stop`
Les fichiers `S` s'exécutent au démarrage du système dans l'ordre de `nn`. Par exemple, `S01commande1` s'exécutera avant `S55commande2`.

A l'arrêt, ce sont les fichiers K qui s'exécutent selon le même principe.

```
# ls rc6.d
K01kdm          K20makedev            K79nfs-common     S35networking
K11atd          K20openbsd-inetd      K89klogd          S36ifupdown
K11cron         K20pure-ftpd          K90sysklogd       S40umountfs
K20acpid        K20rsync              README            S50lvm
K20cupsys       K20ssh                S15wpa-ifupdown   S60umountroot
K20dbus         K20xpilot-ng-server   S20sendsigs       S90reboot
K20dirmngr      K23ntp                S30urandom
K20exim4        K25hwclock.sh         S31umountnfs.sh
K20lpd          K50alsa-utils         S32portmap
```

La plupart des logiciels sont livrés avec les procédures permettant de les insérer dans ce schéma de démarrage. Si vous êtes amenés à installer une application développée par vous même, il vous faudra écrire le shell correspondant et décrire les fichiers de démarrage et d'arrêt.

Exemple
Consulter sur votre système l'organisation des commandes lancées au démarrage du système.

8.2 Programmation système

Dans cette section, nous introduisons la programmation système en langage C. Cette introduction a pour but de vous montrer comment les fonctions vues précédemment dans le cours sont programmées. Nous présentons dans cette section : comment passer des paramètres au lancement d'un programme, comment manipuler des fichiers et enfin comment créer un processus.

Vous trouverez en annexe un résumé de cours sur le langage C vous permettant de comprendre les programmes présentés dans la section, de les compiler et de les exécuter.

8.2.1 Passage de paramètres dans la fonction main()

Il est possible de passer des paramètres à un programme écrit en C. Ces paramètres sont transmis par le shell de la façon suivante :

```
int main(int argc, char* argv[], char* envp[])
```

où :

⇒ `argc` (argument count) est le nombre d'arguments passés lors de l'appel du programme.

⇒ `argv` (argument vector) est un tableau de chaines de caractères contenant les arguments passés dans la ligne de commande, ils sont découpés en mots par le

shell (chaînes de caractères). Chaque élément du tableau pointe sur une chaîne de caractères. `argv[0]` contient le nom de la commande.

⇒ `envp` (environment pointeur) un tableau de chaines de caractères qui contient les variables d'environnement du shell au moment de l'exécution au programme.

ExempleMain Ecrire un programme C qui affiche : le nom du programme, la liste des paramètres passés au programme (s'il y en a) et les 10 premières variables d'environnement.

```
1  #include <stdio.h> /* pour les entrees-sorties */
2
3  int main(int nbarg, char* argv[], char* envp[])
4  {
5    int i;
6
7  printf("Nom du programme: %s \n",argv[0]);
8  printf("Nombre d'arguments: %d \n",nbarg);
9  printf("\n");
10 printf("Liste des parametres: \n");
11 if (nbarg>1)
12   {
13     for (i=1;i<nbarg;i++)
14       {
15         printf("parametre no%d: %s \n", i, argv[i]);
16       }
17   } else
18       {
19         printf("il n'y pas de parametre \n");
20       }
21 printf("\n");
22 printf("Liste des variables d'environnement: \n",argv[0]);
23 for (i=1;i<=10;i++)
24   {
25   printf("env no%d: %s \n", i, envp[i]);
26   }
27 return 0;
28 }
```

8.2.2 Accès aux fichiers Unix

Il n'y a pas de commandes standards, un fichier est une suite de caractères. L'accès aux fichiers est direct.

On inclut les fichiers `<sys/file.h>` et `<sys/stat.h>` pour pouvoir utiliser les fonctions d'accès aux fichiers et les constantes correspondantes. Ces fonctions utilisent un *file descriptor* ou *descripteur de fichiers* pour référencer un fichier. Ce descripteur de fichiers correspond à l'index du fichier dans la table des fichiers ouverts du processus. On rappelle que dans cette table les index 0, 1 et 2 sont réservés respectivement pour l'entrée standard, la sortie standard et l'erreur.

Création de fichier

est réalisée à l'aide de la fonction `creat()`. Si le fichier existe déjà il est remis à 0 : l'ancien est écrasé.

```
int fd = creat(char* name, int perms);
```

`fd` : est le descripteur de fichier rendu par la fonction `creat()`, s'il y a eu une erreur lors de la crétion fd vaut -1,

`name` : est le nom du fichier,

`perms` : sont les droits d'accès au fichier. Le `umask` du processus est appliqué à cette valeur `perms`. Le fichier sera donc créé avec les droits : `perms & ~umask`

Listing 8.5– Exemple de création de fichier

```
1  #include <stdio.h> /* pour les entrees-sorties */
2  #include <sys/file.h> /* pour les primitives de manipulation de fichiers
       */
3  #include <sys/stat.h> /* pour les permissions */
4
5  int main()
6  {
7    int fd1;
8    char* nom;
9    int perms;
10
11   nom="fichCreat";
12   perms=0777;
13   fd1 = creat(nom,perms);
14   printf("le descripteur de fichier1 est %d  \n",fd1);
15   return 0;
16   }
```

Ouverture de fichier

Un fichier doit toujours être ouvert afin de pouvoir travailler dessus et un fichier doit être créé pour pouvoir être ouvert.

```
int fd = open(char* name, flags, int perms);
```

`fd` : descripteur de fichier

`name` : nom du fichier,

`flags` : mode d'ouverture du fichier.

`perms` : droits d'accès au fichier (seulement en cas de création).

Les différents modes d'ouverture sont :

⇒ O_RDONLY : lecture seule.
⇒ O_WRONLY : écriture seule
⇒ O_RDWD : lecture et écriture
⇒ O_CREAT : création du fichier s'il n'existe pas
⇒ O_APPEND : ouverture du fichier en mode ajout, initialement et avant chaque écriture, la tête de lecture/écriture est placée à la fin du fichier.

Différents modes peuvent être précisés, séparés par des `|`

Exemple :

```
int fd = open("fichCreat", O_RDONLY);
```

Lecture de fichier

```
int nb = read(int fd, char* buf, int size);
```

`nb` : nombre de caractères lus (= -1 s'il y a une erreur).

`fd` : descripteur de fichier (obtenu via la fonction `open`).

`buf` : adresse du buffer où stocker les données lues.

`size` : taille des données à lire.

Listing 8.6– Exemple de lecture de fichier

```
nb = read(fd, bufRead, 4);
```

Ecriture dans un fichier

```
int nb = write(int fd, char* buf, int size);
```

`nb` : nombre de caractères écrits

`fd` : descripteur de fichier

`buf` : adresse du buffer contenant les données à écrire

`size` : taille des données à écrire

Fermeture d'un fichier

```
int err = close(int fd);
```

`fd` : descripteur du fichier à fermer

8.2.3 Création de processus

Les fonctions `getpid()` et `getppid()` permettent d'obtenir le `pid` du processus courant et celui de son père.

Listing 8.7– Exemple d'utilisation de `getpid()`/`getppid()`

```
#include <unistd.h>
pid_t pid = getpid(void)
pid_t pid = getppid(void)
```

La fonction `fork` La primitive `fork` permet à un processus (processus père) de créer de façon dynamique un processus (processus fils). Le processus fils s'exécute de façon concurrente avec le processus père. Tout processus Unix (sauf le processus 0) est crée à l'aide de cette primitive.

Listing 8.8– Exemple d'utilisation de `fork()`

```
#include <unistd.h>
pid_t pid = fork(void)
```

Le processus fils est une copie conforme de son processus père : le processus fils exécutera le même programme que son père sur une copie des données de celui-ci au moment de l'appel à la primitive et en particulier une copie de la pile d'exécution. Cependant les variables ne sont pas partagées. Chaque processus a sa propre copie des donnnées et de la pile : si un processus modifie une variable, cela ne modifie pas la variable équivallente de l'autre processus. Egalement, le processus fils héritent des descripteurs de fichiers du processus père : ils partagent alors l'accès aux fichiers ouverts avant le `fork()`.

De cette façon, une fois le processus fils crée, cela se passe comme si le père et le fils avaient fait appel à la primitive `fork()`. Chacun des deux processus reprend son exécution au même point, c'est à dire au retour de l'appel à `fork()`. C'est le code de retour de cette fonction qui distingue le père du fils, cette valeur est :

⇒ `0` dans le processus fils

⇒ le `pid` du processus fils créé dans le processus père

Si la primitive `fork()` échoue, le code de retour est `-1`. Dans ce cas il n'y a pas de nouveau processus de créer. La primitive `fork()` peut échouer lorsque l'utilisateur a créer trop de processus ou si le nombre total de processus dans le système est trop élévé.

La fonction `exit` Elle termine le processus courant. Ses descripteurs de fichiers sont fermés. S'il reste des processus fils non terminés, ceux-ci reçoivent le processus `1` pour

père.

Listing 8.9– Exemple d'utilisation de `exit()`

```
void exit(int status);
```

`status` : code de retour du processus.

La fonction `sleep()` Elle suspend un processus pendant le nombre de secondes indiqué en paramètre.

Listing 8.10– Exemple d'utilisation de `sleep()`

```
unsigned int err = sleep(unsigned int seconds);
```

La fonction `wait()` Elle bloque un processus en attente de la fin de l'exécution d'un processus fils qu'il a créé. Elle ne permet pas d'attendre la fin d'un fils en particulier (`waitpid()` le permet).

Listing 8.11– Exemple d'utilisation de `wait`

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

La fonction retourne le numéro de `PID` du fils qui vient de se terminer, `status` peut contenir des inforamations sur la façon dont s'est terminé le processus fils (valeur du `exit` du fils).

ExempleFork Création d'un nouveau processus par appel à la primitive `fork`. Dans les exercices, à la fin du chapitre nous testerons l'exécution de ce programme.

```c
#include <stdio.h>
#include <unistd.h> /* constante*/
#include <stdlib.h> /* fonction exit */


int main() {
  pid_t pid;

  switch (pid=fork()) {
    case  (pid_t)-1: printf("erreur creation de processus");
      exit(2);
    case (pid_t)0:
      /* on est dans le processus fils */
      printf("valeur de fork= %d ",pid);
      printf("Je suis le processus fils: mon pid= %d, pid de mon
                                %pere=%d\n",getpid(), getppid());
      printf("Fin du processus fils\n");
      exit(0);

    default:
      /* on est dans le processus pere */
      printf("valeur de fork= %d ",pid);
      printf("Je suis le processus pere: mon pid=%d, pid de mon
                                %pere=%d\n",getpid(), getppid());
            printf("Fin du processus pere\n");
            exit(0);
    }

}
```

8.3 Exercices

Exercice 21 : Passage de paramètres dans la fonction main()
Tester le programme `exempleMain` donné ci-dessus.

Exercice 22 : La création de processus

Question 22.1 : Exécuter plusieurs fois le programme `exempleFork` donné au paragraphe précédent. Lors de l'exécution de ce programme, les processus s'exécutent de façon concurrente. Ils peuvent alors avoir des exécutions entrelacées. Expliquer les différents résulats obtenus.

Question 22.2 : Quelles solutions proposez-vous pour permettre au processus fils de terminer avant son père ? Tester.

Question 22.3 : Ajouter les instructions suivantes dans le programme :
- ⇒ Déclarer une variable globale dans le programme.
- ⇒ Initialiser cette variable et afficher sa valeur avant le `fork`.
- ⇒ Incrémenter cette variable de `5` dans le code du processus père et afficher sa valeur.
- ⇒ Incrémenter cette variable de `10` dans le code du processus fils et afficher sa valeur.

Exécuter le programme. Qu'est ce que ce test confirme ?

Exercice 23 : Accès aux fichiers Unix

Question 23.1 : Ecrire un programme C `exempleFich` qui :
- ⇒ ouvre un fichier `fichCreat` en mode création et écriture
- ⇒ écrit la chaîne de caractères "bonjour" dans le fichier `fichCreat`
- ⇒ ouvre le fichier `fichCreat` en mode lecture et ecriture
- ⇒ lit 9 caractères dans le fichier `fichCreat` avec le nouveau descripteur (A)
- ⇒ affiche les caractères lus à l'écran
- ⇒ écrit la chaîne de caractères "la promo" dans le fichier `fichCreat` avec le même descripteur que précédemment (B)
- ⇒ ferme les deux ouvertures de fichiers

Exécuter votre programme. Vérifier qu'un fichier `fichCreat` a bien été créé et qu'il a la taille escomptée.

Question 23.2 : Reprendre le programme `exempleFich` et le modifier en inversant les dernières lectures/écritures (A et B). Que se passe-t-il ? Comment éviter cela ?

Question 23.3 : Ecrire un programme C qui :
- ⇒ ferme le descripteur de l'entrée standard
- ⇒ crée un fichier dont le nom est passé en paramètre du programme
- ⇒ écrit "bonjour" à l'aide de la commande `printf`

Exécuter ce programme. Que se passe-t-il ? Afficher le contenu du fichier créé.

8.4 Correction Exercices

Correction Exercice 21 : La création de processus

Solution question 21.1 : le père et le fils s'exécutant de façon entrelacée, le père peut se terminer avant le fils : dans cas, le fils a pour père le processus 1.

Solution question 21.2 : Ajouter à la fin du code du processus pere soit :

⇒ la fonction `wait()` qui permet au processus père d'attendre la fin de l'un de ses fils.

⇒ la fonction `sleep(sec)` qui permet au processus père d'attendre `sec` secondes. Ainsi, le processus fils a le temps de terminer.

```
1  #include <stdio.h>
2  #include <unistd.h> /* constante*/
3  #include <stdlib.h> /* fonction exit */
4
5
6  int main() {
7    pid_t pid;
8    int var1;
9
10   printf("avant le fork, valeur de var1 %d\n",var1);
11
12   switch (pid=fork()) {
13     case  (pid_t)-1: printf("erreur creation de processus");
14       return(2);
15     case (pid_t)0:
16       /* on est dans le processus fils */
17       printf("valeur de fork= %d ",pid);
18       printf("Je suis le processus fils: mon pid= %d, pid de mon
19                                  %pere=%d\n",getpid(), getppid());
20       printf("Fin du processus fils\n");
21       return(0);
22
23     default:
24       /* on est dans le processus pere */
25       printf("valeur de fork= %d ",pid);
26       printf("Je suis le processus pere: mon pid=%d, pid de mon
27                                  %pere=%d\n",getpid(), getppid());
28       wait();
29       printf("Fin du processus pere\n");
30             return(0);
31     }
32
33 }
```

Solution question 21.3 : Ce test permet de montrer que le processus fils hérite d'une copie des variables de son père. Chaque processus travaille sur sa propre copie.

```c
#include <stdio.h>
#include <unistd.h> /* constante*/
#include <stdlib.h> /* fonction exit */


int main() {
  pid_t pid;
  int var1;

  var1=3;
  printf("avant le fork, valeur de var1 %d\n",var1);

  switch (pid=fork()) {
    case  (pid_t)-1: printf("erreur creation de processus");
      return 2;
    case (pid_t)0:
      /* on est dans le processus fils */
      printf("valeur de fork= %d ",pid);
      printf("Je suis le processus fils: mon pid= %d, pid de mon
                                %pere=%d\n",getpid(), getppid());
      printf("Je suis le processus fils: var1=%d\n",var1);
      var1=var1+10;
      printf("Je suis le processus fils: var1=%d apres incrementation \n"
          ,var1);
      printf("Fin du processus fils\n");
      return 0;

    default:
      /* on est dans le processus pere */
      printf("valeur de fork= %d ",pid);
      printf("Je suis le processus pere: mon pid=%d, pid de mon
                                %pere=%d\n",getpid(), getppid());
      printf("Je suis le processus pere: var1=%d\n",var1);
      var1=var1+5;
      printf("Je suis le processus pere: var1=%d apres incrementation \n"
          ,var1);
      wait();
      printf("Fin du processus pere\n");
            return 0;
    }

}
```

Correction Exercice 22 : Accès aux fichiers Unix

Solution question 22.1 : L'exécution de ce programme crée un fichier `FichCreat`, dans lequel il y a la chaîne "bonjour la promo". On peut vérifier cela avec la commande `ls -l`.

Programme `exempleFich` :

```
1  #include <stdio.h>
2  #include <sys/file.h> /* pour les primitives de manipulation de fichiers
       */
3  #include <sys/stat.h> /* pour les permissions */
4
5  int main()
6  {
7    int fdCreat,fdWrite, fdRW;
8    int nbEcrit, nbLu;
9    int err;
10   char* bufW;
11   char bufR[20];
12
13   /* Ouverture du fichier en mode creation et ecriture */
14   fdWrite= open("fichCreat", O_WRONLY|O_CREAT, 0777);
15   printf("Descripteur fdWrite est: %d  \n",fdWrite);
16
17   /* Ecriture de 9 caractères dans le fichier */
18   bufW = "bonjour";
19   nbEcrit = write(fdWrite, bufW, 9);
20   printf("le nombre de caracteres ecrits est %d \n",nbEcrit);
21
22   /* Ouverture du fichier en mode lecture et ecriture*/
23   /* Offset repositionne en debut de fichier */
24   fdRW= open("fichCreat", O_RDWR, 0);
25   printf("Descripteur fdRW est: %d  \n",fdRW);
26
27   /* Lecture de 9 caractères dans le fichier */
28   /* Cela positionne l'offset en fin de fichier */
29   nbLu = read(fdRW, bufR, 9);
30   printf("le nombre de caracteres lus est %d \n",nbLu);
31   printf("la chaine lue est %s \n", bufR);
32
33   /* Ecriture de 9 caractères a la suite dans le fichier */
34   bufW = "la promo";
35   nbEcrit = write(fdRW, bufW, 8);
36   printf("le nombre de caracteres ecrits est %d \n",nbEcrit);
37
38   /* Fermeture des deux descripteurs de fichiers */
39   err = close(fdWrite);
40   printf("fermeture de fdWrite, code erreur: %d\n", err);
41
42   err = close(fdRW);
43   printf("fermeture de fdRW, code erreur: %d\n", err);
44
45   return 0;
46 }
```

Solution question 22.2 : Une fois que l'on a ouvert le fichier en lecture/écriture :

⇒ on commence par écrire : on écrase ce qu'il y avait dans le fichier car l'ouverture a repositionné l'offset au début du fichier (Le vérifier avec la commande ls -l).

Pour éviter cela, il faut ajouter le mode `APPEND` à l'ouverture qui permet de positionner l'offset à la fin du fichier. (Le vérifier avec la commande ls -l)

⇒ puis on lit : la lecture rend 0 car l'offset est positionné à la fin du fichier après l'écriture. Pour solutionner cela, on peut ouvrir à nouveau le fichier, ce qui positionnera l'offset au début du fichier.

Solution question 22.3 : Comme le descripteur correspondant à la sortie standard est fermé, il n'y a plus d'affichage à l'écran. Lorsque l'on crée un fichier le premier descripteur de fichier libre est attribué. Dans ce cas, lors de la création du fichier, le descripteur numéro `1`, correspondant à la sortie standard est attribué. De ce fait, lorsqu'un `printf` est réalisé, l'écriture ce fait dans le fichier nouvellement créé.

```
1  #include <stdio.h>
2  #include <sys/file.h> /* pour les primitives de manipulation de fichiers
       */
3  #include <sys/stat.h> /* pour les permissions */
4
5  int main(int nbarg, char* argv[], char* envp[])
6  {
7    int fdCreat;
8
9    /* Fermeture de la sortie standard */
10   close(1);
11
12   /* Creation d'un fichier dont le nom est passe en parametre */
13   fdCreat = creat(argv[1],0777);
14   printf("le descripteur de fichier3 est %d  \n",fdCreat);
15
16   /* Affichage de bonjour sur la sortie standard */
17   printf("bonjour \n");
18
19   return 0;
20
21 }
```

Deuxième partie

Annexes

Chapitre 9

Liste des principales commandes

La liste des principales commandes Unix **dont vous disposerez à l'examen** se trouve sur les deux pages suivantes.

Commandes	**Description**	**Exemples**
at	exécution d'une commande à l'heure donnée	at 1023
cal	donne le calendrier	cal 1998 cal 7 1997
cat	affiche le contenu d'un fichier -n : précède chaque ligne par son numéro	cat fich cat -n fich
cd	changement de répertoire	cd rep
chmod	change les droits d'accès d'un fichier -R : changement récursifs aux sous-rép.	chmod u+x fich chmod -R a+x rep
comm	affiche les lignes communes à deux fichiers	comm fich1 fich2
cp	recopie un fichier -r : recopie de répertoire -i : demande avant d'effectuer une recopie	cp path1 path2 cp -r rep1 rep2 cp -i fich1 fich2
cut	affiche une partie de chaque ligne d'un fichier -c : par numéros de colonnes (1 à 3 et 8) -f : par champs séparés par des délimiteurs -d*c* : où *c* donne le caractère délimiteur	 cut -c1-3,8 fich cut -f1,2 fich cut -f1 -d. fich
date	affiche la date courante	date
df	donne l'espace libre dans les systèmes de fichiers	df
du	donne le nombre de bloc disque utilisé par un fichier ou un répertoire	 du rep
diff	donne les différences entre deux fichiers -i : majuscules = minuscules	diff fich1 fich2
echo	affiche une chaîne -n : affiche la chaîne sans nouvelle ligne	echo blabla echo -n blabla
exit	fin de session (ou logout)	
find	recherche d'un fichier dans l'arborescence -name : précise un fichier particulier -print : affiche le chemin d'accès	find path find path -name fich1 find / -name mf2 -print
grep	recherche une chaîne dans un fichier -i : cherche la chaine en maj. et min. -n : numérote les lignes -v : sélectionne les lignes ne contenant pas chaîne -w : sélectionne les lignes ou chaîne est un mot	grep chaine fich grep -ichaine fich
head	affiche les 10 premières lignes d'un fichier	head fich
hostname	donne le nom de la machine courante	
kill	tue un processus	kill pid
lpr	imprimer	
ls	liste le contenu du répertoire -a : affiche aussi les fichiers commençants par . -R : affiche aussi le contenu des sous-répertoires -i : donne le numéro d'inode	ls rep ls -a ls -R ls -i fich
man	aide en ligne des commandes	man man
mesg	permet ou refuse l'affichage de messages	mesg y [n]
mkdir	création d'un sous-répertoire	mkdir rep
more	affiche un texte page par page	more fich

Commandes	**Description**	**Exemples**
mv	renomme un fichier ou répertoire	mv fich1 fich2
nl	numérote les lignes d'un fichier	nl fich
nice	modifie la priorité d'un processus	nice +1 prgm
passwd	permet de changer le mot de passe (yppasswd)	passwd user
ps	liste les processus -a : donne le liste de tous les processus -u : donne le nom des propriétaires	
pwd	affiche le nom du répertoire courant	
rm	efface un fichier -r : efface un répertoire (ou rmdir si répertoire vide) -f : force en cas de droits d'accès limités -i : demande confirmation	rm fich rm -r rep rm -rf rep rm -ri rep
ssh	connexion à distance (ou telnet)	ssh smith
sleep	attente d'un délais	sleep 10
sort	tri d'un fichier, basé sur les codes ASCII -r : tri inversé -d : tri uniquement sur les caractères, chiffres et espaces -i : ne trie que les codes ASCII entre 32 et 126 -b : sans différence entre min. et maj.	sort fich
tail	donne les dernières lignes d'un fichier	tail fich
tee	écrit sur un fichier et sur l'écran dans une suite de pipe	cat fich1 |tee fich2
time	donne les temps d'exécution d'une commande	time du -s
tr	modification de chaînes de caractères -d : efface les caractères donnés	tr ch1 ch2 tr -d xyw
tty wall	donne l'identificateur du terminal envoie un message a tout le monde	tty wall fich
wc	donne le nombre d'octets, mots et lignes d'un fichier -l : uniquement le nombre de lignes -w : uniquement le nombre de mots -c : uniquement le nombre d'octets	wc fich wc -lc fich
who	donne les utilisateurs connectés	
;	séparateur de commandes	date ; who
(commande)	séquence de commandes qui n'affectent pas le niveau courant	(cd /etc ; ls)
<	redirection de l'entrée depuis un fichier	read a < fich
>	redirection de la sortie sur un fichier	date > sortie
>>	ajoute la sortie à la fin du fichier	
|	utilise la sortie de la première commande comme entrée de la seconde	ls | more
&	exécute une commande en arrière plan	

Chapitre 10

Interprétation des caractères spéciaux

	Génération de noms de fichiers	**Expressions régulières**
?	un caractère quelconque, sauf <new line>	remplace 0 ou 1 fois le caractère qui précède
.	le caractère point	un caractère quelconque sauf <new line>
*	0 ou un nombre quelconque de caractères	remplace 0 fois ou n fois le caractère qui précède
`[a-i]`	un caractère entre a et i	un caractère entre a et i
`[!a-i]`	un caractère qui n'est pas entre a et i	un caractère entre a et i, ou !
`[^a-i]`	un caractère qui n'est pas entre a et i	un caractère qui n'est pas entre a et i
`\`	banalise le caractère qui suit	banalise le caractère qui suit
^	le caractère ^	ce qui suit est en début de ligne
$	référence une variable	ce qui précède est en fin de ligne

Chapitre 11

Le langage C

Ce chapitre est une introduction à la programmation en langage C qui vise à vous aider à commencer la programmation système. Il ne s'agit pas d'un cours, mais de notes de cours, ce qui explique qu'il n'est pas rédigé.

11.1 Introduction

C'est un langage "système" qui a été développé pour écrire Unix.

Il est structuré et permet de descendre à bas niveau. Il permet la gestion des accès mémoire et a une interface facile avec l'assembleur.

D'un certain côté, il s'agit d'un langage facile au niveau de sa syntaxe mais :
⇒ complexe au niveau de la compilation, les compilateurs sont laxistes au niveau des contrôles de type. Il permet la conversion de type.
⇒ on doit savoir ce qui se passe au niveau de la mémoire pour programmer en C.
⇒ il n'y a pas de type pointeur.

11.2 Structure d'un programme

Exemple de programme C :

```
int a;

main()
{
  int i;

/* ceci est un programme essai */

  i=4;
...

}
```

Un programme C a la structure suivante :

```
variables

fonctions dont  la fonction principale qui s'appelle main( ) (point
d'entrée du pg)

blocs
```

Toute ligne du programme est terminée par un point virgule.

11.2.1 Définition et déclaration des variables

Lors de la déclaration, l'emplacement est réservé en mémoire.

```
type var1, var2, ...;
```

11.2.2 Blocs

Ils sont délimités par des accolades `{}`.

11.2.3 Déclaration des fonctions

```
type nomfonction(type1 vara, type2 varb, ...) {
  ...
  return valeur;
}
```

Remarques :

⇒ la valeur retournée par la fonction doit être du type donné dans l'entête de cette fonction.
⇒ la fonction principale main( ) se déclare de la même façon.
Exemple :

```
main()
{
   printf("hello world");
}
```

11.2.4 Appel de fonctions

```
val=nomfonction(var1, var2, ...);
```

11.3 Types de données

11.3.1 Types prédéfinis

char : caractères sur 1 octet

int : entiers sur 4 octets

short : entier sur 2 octets.

long : entier sur 4 octets.

float : virgule flotante sur 4 octets.

void : type générique utlisé pour les fonctions ne retournant pas de valeur. Dans ce cas, il n'y a pas besoin de mettre de *return* ou alors *return* sans paramètre.

Exemple :

```
char lettre1, lettre2;
int i;
```

```
void afficheCar(char c){
    printf("c = \%c \n", c);
}
```

11.3.2 Types composés

Définition/Redéfinition de type

est réalisé grâce à `typedef`.

Exemple :

```
typedef int longueur; /* définition d'un type longueur */

longueur l; /* déclaration de l de type longueur*/

main()
{
l = 300; /* utilisation de la variable l*/
}
```

Définition de structures de données

On peut définir des structures de données grâce à `typedef`

```
typedef struct {  /* définition du type Tpoint */
  int x;
  int y;
}Tpoint;

Tpoint p1; /* déclaration de p1 de typt Tpoint*/

main() {
  p1.x = 100; /* utilisation de p1*/
  p1.y = 200;
}
```

Type énuméré

On peut se définir un type énuméré à partir de `enum` qui permet de déclarer des "constantes énumérées".

Exemple :

```
typedef enum {vrai,faux} Tboolean; /* définition du type
   Tboolean*/
Tbolean exist; /* une variable de type Tboolean */
```

Par défaut, le premier élément à l'intérieur des acolades vaut 0, le second 1, etc. On peut imposer des valeurs avec le signe =.

```
typedef enum {rouge, bleu, vert=5, violet} Tcouleur;
```

où `rouge` vaut 0, bleu 1, violet 6

Les tableaux

Il n'existe pas de type tableaux. Pour définir un tableau :

```
int tab[10];
```

Pour initialiser un élément du tableau :

```
tab[4] = 50;
```

Un tableau est une suite de variables, adressées à partir de l'adresse d'origine. Il est indicé de 0 à n-1.

Remarques :
La variable *tab* ne contient pas un tableau, mais l'adresse du début du tableau. Pour accéder à tab[4], on calcule l'adresse du 4ème élément : $tab + 4 * sizeof(int)$.

Les structures

c'est une ancienne possibilité conservée pour des raisons de compatibilité.

```
/* Définition de la structure point */
struct point {
  int x;
  int y;
};

struct point p1; /* déclaration de p1*/

main()
{
  p1.x = 100; /* utilisation de p1*/
  p1.y = 200;
}
```

11.4 Déclaration de variables

11.4.1 Variables locales

⇒ Elles sont définies dans un bloc, et cela au début de n'importe quel bloc.
On peut, par exemple, définir une variable locale à une structure for.
Cette utilisation n'est pas conseillée car elle n'est pas très lisible.

⇒ Ces variables locales sont définies sur la pile.

Exemple :

```
int fonction()
{
  int i,j;

  i=4;

  return i;
}
```

11.4.2 Variables globales

Elles sont déclarées en dehors de la définition des fonctions. Elles sont définies dans les données du programme (du binaire exécutable).

Exemple :

```
int nbEtud;
main(){
 int i,j;

  i=4;
  nbEtud=5;

}
```

11.4.3 Initialisation

Les variables peuvent être initialisées lors de leur déclaration (définition), mais ceci n'est pas recommandé.

Par défaut les variables globales sont initialisées à 0 (BSS).

11.4.4 Variables externes

Elles sont définies dans un autre fichier et utilisées en local. Elles permettent de faire connaître des variables d'une fonction à une autre.

Exemple :

```
extern int toto; /* on ne réserve pas de place car variable
   déjà
                    définie */
```

et dans un autre fichier on trouve :

```
int toto
```

11.5 Entrée/Sorties

11.5.1 Affichage

Est réalisé par la fonction `printf` qui est définie dans le fichier `stdio.h`.

```
printf("chaine + format", variables);
```

Cette fonction permet d'afficher une chaîne de caractères si on le désire, et les valeurs des variables. On doit préciser le format des variables dont on veut afficher la valeur.

Exemple :

```
#include <stdio.h>

main(){
  int i;
  char car;

  i=4;
  car= 'a';
  printf("caractère = %c, entier=%d \n", car, i);
}
```

donne à l'écran :

```
caractère=a , entier=4
```

Les différents formats :

⇒ %d : entier
⇒ %c : caractère
⇒ %s : chaîne de caractères
⇒ %f : flottant
⇒ %x : hexa

Les différents contrôles :

⇒ n : retour à la ligne
⇒ t : tabulation
⇒ 0 : nul (fin de chaîne)

11.5.2 Saisie

Est réalisée par la fonction `scanf` qui s'utilise de la façon suivante :

```
scanf("format", arg1, arg2, ...)
```

Le format est le même que pour l'affichage. Par contre, on ne peut pas écrire de chaîne de caractères et en ce qui concerne les variables saisies, on passe leurs adresses en argument (&).

Exemple :

```
printf("saisir c et i \n");
scanf("%c, %d \n", &car, &i);
```

11.6 Opérateurs

11.6.1 L'opérateur d'affectation :

=

11.6.2 Les opérateurs arithmétiques :

+, −, *, $/

11.6.3 Les opérateurs relationnels et logiques :

Les opérateurs relationnels :

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

Les opérateurs logiques :

```
&&, ||
```

Les opérateurs arithmétiques ont une précédence plus forte que les opérateurs relationnels.

Par contre, pour les expressions contenant des opérateurs logiques, elles sont évaluées de gauche à droite et l'évaluation est arrêtée dés qu'un résultat est trouvé (juste ou faux).

11.6.4 Les opérateurs d'incrémentation :

++, –

Exemple :

```
i++; /* correspond à i=i+1 ou i+=1 */
```

Attention :

++i != i++

```
i=0;
a=i++;
```

donne a = 0 et i=1
car cela se traduit par :

```
i=0;
a=i;
i=i+1;
```

Par contre :

```
i=0;
b=++i;
```

donne b = 1
car cela se traduit par :

```
i=0;
i=i+1;
b=i;
```

11.6.5 L'opérateur adresse :

`&`

Exemple :

```
int i;  /* réservation de 4 octets en mémoire */
ad = &i; /* adresse mémoire des 4 octets */
i=2;
```

11.6.6 L'opérateur contenu d'adresse :

`*`

`*(&i)` donne i, c'est à dire le contenu de i

11.6.7 Les opérateurs binaires

Le C offre 6 opérateurs pour manipuler les bits :

⇒ Le ET (`&`) positionne le bit à 1 si les bits sont à 1 dans les deux opérandes et à 0 sinon.

⇒ Le OU (`|`) est utilisé pour positionner des bits à 1, par exemple :

```
x= x | SetOn
```

met à 1 les bits de x qui sont à 1 dans SetOn.

⇒ Le OU exclusif (`^`) met les bits à 1 quand ils sont différents dans les deux opérandes et à 0 quand ils sont identiques.

⇒ Les opérateurs de décalage à gauche (<<)et à droite (>>)opèrent un décalage sur l'opérande qui est située à leur gauche du nombre de positions donné par l'opérande de droite (ex : `x<<2`). Les bits qui sont libérés sont remplis par des 0.

⇒ L'opérateur complément unaire (~) convertit chaque 1-bit en 0-bit et vice versa.

11.7 Structures de contrôle

11.7.1 If

```
If (condition) {
   instructions;
} else {
   instructions;
}
```

Exemple :

```
 if (codeErr == 0) {
   printf("Résultat OK");
 }
 else {
   printf("Erreur");
 }
```

11.7.2 For

```
for (initialisation; condition d'arrêt; incrément) {
    instructions;
}
```

Exemple :

```
for (i=0; i<10; i++) {
    tableau[i] = 4 * i;
}
```

11.7.3 While

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

Exemple :

```
i=0;
while (i<10) {
    printf(``i= %d \n'',i);
    i=i+1;
}
```

11.7.4 Répéter

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

Exemple :

```
int n;
do {
   printf("Donner un nbre >0 \n");
   scanf("%d",&n);
} while (n<=0);
printf("Réponse correcte");
```

11.7.5 Cas

Quand on entre dans une branche du case, on exécute tout ce qui suit jusqu'à un break.

```
switch (valeur){
       case cas1:instruction1;
                 instruction;
                 ...
                 break;
        case cas2:instruction1;
        case cas3:
        case cas4:instruction2;
                  break;
        default: instruction;
}
```

Les arguments du case sont soient des caractères, soient des entiers.

Explications : Pour cas1, on exécute les instructions données dans cas1 et on sort au `break`.

Pour cas2, on exécute instruction1 et instruction2 car il n'y a pas de break (dans ce cas, quand on entre dans un cas, on exécute tout ce qui suit jusqu'à un break.)

Pour cas3, on exécute instruction2.

Pour cas4, instruction2.

Pour default, on exécute cette séquence si on a rien exécuter jusque là. On n'a pas besoin de break.

Exemple : Exemple d'utilisation de la structure `case` :

```
variable char n;
switch (n){
        case 'y':reponse_oui();
                 break;
        case 'n':reponse_non();
                 break;
        default: printf("erreur \n");
}
```

11.7.6 Les ruptures de séquences

`break` : permet de sortir d'une boucle.

`continue` passe à l'itération suivante.

11.7.7 Conditions

Elles peuvent contenir :

⇒ Une comparaison :
`i<10 ou b>a ou i!=j ou r==20`

⇒ Une variable :

```
if (a)  si a=0 faux, sinon vrai (vrai si a!=0)
```

⇒ Un appel de fonction : on peut utiliser les deux versions, mais la première est conseillée :

```
if (fonc(a)!=0)
if (fonc(a))
```

11.8 Pointeurs

Un pointeur en C est une adresse mémoire, câd 4 octets qui si ils sont initialisés, mémorisent une adresse mémoire.

Déclaration : Il n'y a pas de type pointeur.

```
typeA *nom-variable; /* nom-variable est l'adresse d'un
   élément de type typeA */
```

Attention ! la déclaration d'une variable pointeur permet de réserver la place mémoire nécessaire à la mémorisation d'une adresse : 4 octets et ceci quelque soit le type pointé.

Exemple :

```
int *Pint; /* Pint est un pointeur sur un entier */
```

Utilisation : *Pint* est une adresse (celle d'un entier) et **Pint* est le contenu de la mémoire située à cette adresse.

Exemple :

```
int i;
int *Pint;
Pint = &i;
i=4;
printf("valeur = %d ", *Pint); /* ceci va afficher 4:*Pint
   est égal à */
                                /* *(&i),  donc le contenu de
                                    l'adresse */
```

Remarques :
Les pointeurs sont des valeurs entières. On peut leur appliquer des opérateurs.

Incrémentation de pointeurs : elle est fonction du type pointé.

Exemple :

```
char *pChar;
int *Pint;
```

pChar ++ donne pChar + 1 (car pointeur sur un caractère)

Pint ++ donne Pint + 4 (car Pint est un pointeur sur un entier)

Pint + 1 donne Pint + 1

Déclaration de tableaux : se fait de la façon suivante :

```
int table[10];

pour accéder à table[5]: *table + 5*sizeof(int)
```

table contient l'adresse de début du tableau, c'est un pointeur sur un int (int *).

Cette déclaration permet de réserver la place mémoire pour ce tableau (10*taille d'un entier).

11.9 Les chaînes de caractères

Il n'y a pas de type chaîne défini dans la norme ANSI.

Une chaîne de caractères est une suite de caractères (tableau) terminée par 0. Cette suite est identifiée par un pointeur.

Exemple :

```
char chaine1[20]; /* défini la place mémoire pour mémoriser
   une chaîne) */

char *chaine2 = "hello world"; /* initialisation de chaine2
   */
```

Par contre, on n'a pas le droit de faire `chaine1="Bonjour";`

On initialise chaine1 caractère par caractère :

```
chaine1[1]= 'b';
chaine1[2]= 'o';
...
chaine1[9]= '\0';
```

ou alors avec `scanf` :

```
scanf(``%s'', chaine1);
```

On peut utiliser chaîne2 :

```
printf("chaine2: %s",chaine2);
```

ou :

```
for (i=0 to ..){
   printf("chaine2[%d]=%c \n",i,chaine2[i]);
}
```

Remarques :

⇒ Il n'y a pas de comparaison ou d'affectation directes.

```
chaine1 == chaine2 /* dans ce cas, le test est
   uniquement réalisé sur
                        les pointeurs cad les adresses */
```

⇒ Par contre, dans `string.h`, il y a des fonctions de bibliothèque qui permettent la manipulation de chaînes de caractères (`strncpy(), strncat(), strncmp()` ...).

11.10 Passage de paramètres dans les fonctions

Il y a deux manières de passer des paramètres :

⇒ par valeur : dans ce cas, il n'y a pas de modification de la variable lors de l'appel de fonction.

⇒ par adresse : il peut y avoir modification de la variable lors de l'appel de fonction.

Exemple :

```
void mafonct(int i, int* j) {
   i = i + 1;
   *j = *j + 2;
   return;
}

main() {
  int a;
  int b;

  a = 1;
  b = 2;
  mafonct(a,&b); /* cô b est passé par adresse, il est
     modifié */
  printf("a= %d, b = %d", a, b);
}
Cela donne: a=1 et b=4
```

11.11 Conversion de type ou casting

Il est possible d'affecter des variables de type différent avec l'opérateur `cast` : `(nom de type)` : pour cela il faut que leurs représentations mémoire soient compa-

tibles.

Exemple :

```
char c = 'o';
int i;

i = (int) c; /* affecte le code ASCII de c à i */
```

`(int)` est une conversion de type ou casting.

L'opération inverse n'est pas possible car `int` est plus grand que `char`.

Ceci est particulièrement utile pour les pointeurs. Quelque soit le type de pointeurs, il est toujours codé sur 4 octets : le casting est donc toujours possible.

Exemple :

```
char nom[8];
int* b;
b = (int*)nom;
```

Dans ce cas, dans le tableau *nom* qui a une taille de 8 caractères (=8*1octet), on pourra mettre 8 div taille(entier) = 8 div 4 = 2 entiers.

11.12 Compilation de programmes C

11.12.1 Compilation classique

permet de générer un exécutable :

```
cc pgm.c -o prgm
```

où :
`pgm.c` est le programme (fichier) source
`prgm` est le programme (fichier) exécutable généré par la compilation, il s'agit de code binaire.
`-o` permet de donner un nom au fichier exécutable. C'est une option : par défault il y a création d'un fichier `a.out`.

11.12.2 Compilation assembleur

permet de générer un programme assembleur.

```
cc pgm.c -s prgm.s
```

Dans ce cas, le fichier résultat ne peut pas être exécuté tel quel, il faut à nouveau le compiler.

11.12.3 Différentes phases d'une compilation

⇒ `pgm.c` : programme source
⇒ `cpp = précompilateur`. Il traite le fichier source avnt le compilateur, manipulateur de chaînes de caractères, il enlève les commentaires, traite les options de précompilation (#).
⇒ `cc= compilateur objet`. Le compilateur lit le programme source une seule fois du début à la fin : les variables ou fonctions doivent donc être définies avant d'être utilisées.
⇒ `ln = édition de liens` : avec les bibliothèques standards et aussi entre fichiers d'un même programme.
⇒ exécutable

11.12.4 Programmes composés de plusieurs fichiers

Si le programme est composé de plusieurs fichiers, il faudra alors en tenir compte lors de la compilation.

Par exemple, si on a deux fichiers source : `pp.c` (programme principal) et `sp.c` (sous programme). Il faut générer les programmes objets et en faire l'édition de liens. Ce qui donne :

```
cc sp.c -o sp.o
cc pp.c -o pp sp.o
```

11.13 Notion de gestion de projets

Lorsqu'un programme devient trop gros, on le découpe en plusieurs fichiers.

11.13.1 Fichiers d'entête ou header (.h)

Déclaration et utilisation Dans les fichiers d'entête on met :

⇒ les déclarations de variables utilisées ensuite en externe,
⇒ les déclarations de types,
⇒ déclaration de constantes,
⇒ fonctions.

Lorsque l'on veut utliser ces fichiers dans un programme, cad dans un *fichier.c*, on utilise *#include nomfich.h* au début du fichier C.

Exemple :
Dans le fichier d'entête `point.h` :

```
typedef struct {
  int x;
  int y;
}TPoint;
```

Dans le programme C utilisant ce type :

```
#include "point.h"
```

Les fichiers d'`include` sont expandés dans les fichiers sources. Il faut faire attention à la déclaration des variables dans les fichiers d'entête pour éviter les duplications.

Exemple :
Dans le fichier d'entête `point.h` :

```
typedef struct {
  int x;
  int y;
}TPoint p;
```

Si `point.h` est inclu dans les fichiers `rond.c` et `carre.c`, alors la variable `p` est déclarée deux fois.

Les fichiers d'entête prédéfinis ils contiennent des fonctions prédéfinies et sont situés sous usr/include. Par exemple :

⇒ `stdio.h` : permet l'utilisation des fonctions standards d'entrée-sortie.
⇒ `string.h` : contient des fonctions de manipulation des chaînes de caractères. ou`strlent` (longueur).
⇒ `math.h` : bibliothèque mathématique.
⇒ `file.h` : accès aux fichiers.

11.13.2 Directives du précompilateur

Elles sont :
⇒ `#include` : inclut les fichiers d'entête
⇒ `#define` et `#undef` : permet de définir des constantes, des états.
⇒ `#ifdef #else #endif` : permet d'éviter des inclusions multiples, des déclarations multiples.
Exemples :

1. Définition de constantes :

```
#define QUATRE 4 /* à la compilation, remplace toutes
   les occurences
de QUATRE par 4 */

#define CHAINE "hello world"
#undef NOM
#define m(x) (128*(x)) /* macro fonction */
```

2. Compilation conditionnelle : dans ce cas, selon si la variable TOTO est définie ou non, on compile une partie du programme ou une autre.

```
#define TOTO

instructions

#ifdef TOTO
   instructions
#else
   instructions
#endif

instructions
```

3. Inclusions multiples : pour éviter les inclusions multiples de fichiers d'entête lorsque l'on a de gros projets :

```
/* au début du fichier point.h */

#ifndef POINT_H
#define POINT_H

 /* tout le point.h */

#endif

/*Ainsi, si le fichier point.h a déjà été inclu, on ne
   le réinclut pas
car POINT_H sera déjà défini.*/
```

11.13.3 L'utilitaire *make*

make est un utilitaire de compilation qui permet de gérer les dépendances entre fichiers : lorsqu'un programme est constitué de plusieurs fichiers source et que l'on fait une modification dans un ou plusieurs de ces fichiers source il n'est pas nécessaire de recompiler l'ensemble des fichiers source, seulement ceux qui ont été modifiés, pour les autres on peut utiliser les fichiers objets pour effectuer l'édition de lien. L'utilitaire *make* compare les dates de dernières modification des fichiers : si un fichier source est plus récent que le fichier objet correspondant lors de l'exécution de *make*, alors il faut recompiler le fichier source.

On utilise un fichier de description des dépendances : *makefile* qui est composé de règles de compilation.

```
target : dépendances
   <TAB> actions
```

où :

target : est le fichier résultat

dépendances : ce sont les fichiers utilisés pour compiler et obtenir la target.

actions : sont des commandes shell, ce sont les lignes de compilation.

Les actions sont appliquées seulement s'il y a eu modification des dépendances depuis la dernière création de target. *make* compare donc les dates de target et des dépendances.

On utilise *maketarget* pour compiler.

Exemple :
On veut compiler un programme pp.c qui utilise un sous programme sp.c. Le programme pp.c inclut un fichier d'entête pp.h et le sous programme sp.c inclut un fichier d'entête sp.h. Le *makefile* correspondant sera :

```
sp.o : sp.c sp.h
       cc sp.c -o sp.o
pp : sp.o pp.c pp.h
       cc sp.c -o pp sp.o
```

On peut définir des variables, par exemple pour donner le chemin d'accès au compilateur :

```
CC = /bin/cc

/* puis utilisation de la variable: */
$(CC) sp.c -o sp.o
```

Bibliographie

[3] **Christian PELISSIER** : *Unix Linux BSD : utilisateurs et développeurs*, HERMES Sciences Publications (2008)

[4] **Andrew Tanenbaum** : *Les systèmes d'exploitation*, Pearson Education France (2008)

[5] **Jean-Michel Lery** : *Unix et Linux : utilisation et administration*, Pearson Education France(2011)

[6] **Jean-Marie Rifflet** : *Unix Programmation et communication*, Dunod (2003)

[7] **JP Armspach, P Colin et F OstréWaerzeggers** : *Linux : initiation et utilisation*, Dunod(2004)

Ce document, produit avec LaTeX 2_ε, a été compilé le 10 octobre 2014.