

---

Module TC-INFO-ASR3

# Comprendre et utiliser un système d'exploitation (à travers Unix et le shell Bash)

---

**Cyril Pain-Barre**

(version du 10/10/2012)



certain droits réservés



# Licence d'utilisation

---

Ce document est assujéti à la licence Creative Commons « *Paternité-Pas d'Utilisation Commerciale-Partage des Conditions Initiales à l'Identique 2.0 France License* » appelée aussi le contrat **by-nc-sa**. Elle est indiquée par la présence de l'un des logos suivants :





ou






Cette licence est décrite sur la page <http://creativecommons.org/licenses/by-nc-sa/2.0/fr/deed.fr> du site de [creativecommons](http://creativecommons.org) et stipule que vous êtes libres :

- **de partager** ce document : le copier, le distribuer ou le transmettre ;
- **de le modifier** afin de l'adapter.

Cependant, ces libertés sont contraintes par les restrictions suivantes :

 ou  **paternité** : le document peut être librement utilisé, à la condition de l'attribuer à son auteur en citant mon nom, mais pas d'une manière qui suggérerait que je vous soutiens ou approuve votre utilisation du document

 ou  **pas d'utilisation commerciale** : vous n'avez pas le droit d'utiliser cette création à des fins commerciales

 **partage à l'identique des conditions initiales** : si vous modifiez, transformez ou adaptez ce document, vous n'avez le droit de distribuer le document qui en résulte que sous un contrat identique à celui-ci.

- Chacune de ces conditions peut être levée si vous obtenez mon autorisation.
- Rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur.

 Ce document est rédigé en L<sup>A</sup>T<sub>E</sub>X. Je n'en communique pas les sources...



# Avant-propos

---

**Ce document est en constante évolution.** Il se veut être un survol des systèmes d'exploitation de la famille **Unix** (et en particulier de **GNU/Linux**), de l'interpréteur de commandes **bash** et de son langage de programmation de scripts. Il est utilisé pour servir de support de cours du module «*Comprendre et utiliser un système d'exploitation*» (TC-INFO-ASR3) du [Programme Pédagogique National du DUT Informatique de septembre 2005](#)<sup>1</sup>. Dans ce cadre, il n'aborde pas la "programmation système" ni le "bas niveau". Certaines commandes ne sont pas étudiées dans cet enseignement. Elles sont présentes à titre informatif ou sont utilisées dans le cadre d'une remise à niveau en entrée d'une Licence Professionnelle DA2I.

Bien qu'assez volumineux et illustré d'un bon nombre d'exemples, il reste une introduction à ce système, néanmoins suffisante pour réaliser des tâches complexes, mais laissant de côté de nombreux détails. Pour des informations plus précises, se reporter à la documentation officielle de la distribution utilisée, ainsi qu'aux manuels en ligne des utilitaires abordés.

Les informations contenues dans ce document n'engagent que moi. De nombreux tests ont été réalisés sur GNU/Linux (distributions Red Hat 6.2 et 7.2, Mandrake 9.0, 9.1, 9.2 et 10.0, Mandriva, Debian Sarge, Etch, Lenny et Sid), et sur Unix SunOS 5.7, afin d'en éprouver le bien fondé.



**Cependant, d'une version à l'autre des utilitaires présentés ici et, *a fortiori*, d'une distribution à l'autre, les résultats des commandes peuvent différer sensiblement.**

Les remarques, les suggestions ou les signalements d'erreurs (!) sont les bienvenus et peuvent être adressés à Cyril Pain-Barre, notamment par e-mail : [cyril.pain-barre@univ-amu.fr](mailto:cyril.pain-barre@univ-amu.fr).

Le lecteur désireux d'en savoir plus pourra se référer à l'un des nombreux ouvrages traitant d'Unix et de bash, parmi lesquels on peut citer :

- Graham Glass, « *Unix for programmers and users* », éditions Prentice Hall
- Matt Welsh, Matthias Kalle Dalheimer et Lar Kaufman, « *Le système Linux* », éditions O'Reilly
- Cameron Newbam et Bill Rosenblatt, « *Le shell bash* », éditions O'Reilly

Version du 10 octobre 2012

---

1. Module de 40 heures.



# Table des matières

---

<b>Licence d'utilisation</b>	<b>iii</b>
<b>Avant-propos</b>	<b>v</b>
<b>Table des matières</b>	<b>vii</b>
<b>1 Prise en mains</b>	<b>1</b>
1.A Introduction	1
1.B Notion de système d'exploitation	1
1.C Unix et GNU/Linux	2
1.D Utilisateurs et groupes Unix	3
1.E Pour les utilisateurs de Microsoft Windows	4
1.F Interpréteur de commandes	4
1.F.1 Terminal	5
1.F.2 Invite de commandes	6
1.F.3 Ligne de commandes	7
1.F.3.a Convention de description d'une commande	7
1.F.3.b Découpage d'une ligne de commande en nom, options et arguments	8
1.F.3.c Raccourci de la description	10
<b>2 Gérer les fichiers</b>	<b>11</b>
2.A Les types de fichiers	11
2.B Organisation des fichiers	12
2.C Références des fichiers	14
2.C.1 Référence absolue et chemin absolu	14
2.C.2 Répertoire de travail, référence relative et chemin relatif	15
2.C.3 Le répertoire . ( <i>point</i> )	15
2.C.4 Le répertoire . . ( <i>point-point</i> )	17
2.D Disques, partitions et systèmes de fichiers	17
2.E Manipulations élémentaires des fichiers	20
2.E.1 cd : changer de répertoire de travail	20
2.E.2 pwd : afficher le répertoire de travail	21
2.E.3 ls : afficher le contenu des répertoires	22
2.E.4 mkdir : créer des répertoires	25
2.E.5 rmdir : supprimer des répertoires	26
2.E.6 cp : copier des fichiers	27
2.E.7 mv : déplacer ou renommer un fichier	29
2.E.8 rm : supprimer des fichiers	30
2.E.9 df : afficher l'occupation disque	31

2.E.10	du : calculer la taille de fichiers ou répertoires	31
2.E.11	quota : connaître la limite autorisée de son occupation disque	32
2.F	Propriété et permissions des fichiers	33
2.F.1	Principe	33
2.F.2	Distinction entre répertoires et fichiers	34
2.F.2.a	Les droits pour un fichier	34
2.F.2.b	Les droits pour un répertoire	34
2.F.3	Connaître ses droits sur un fichier	35
2.F.4	chmod : modifier les permissions de fichiers existants	36
2.F.4.a	Mode absolu	36
2.F.4.b	Mode symbolique	37
2.F.5	umask : enlever des permissions par défaut pour les nouveaux fichiers	39
2.F.6	chgrp : modifier le groupe d'un fichier	41
2.F.7	chown : modifier le propriétaire et le groupe d'un fichier	42
<b>3</b>	<b>Commandes : nature, localisation et aide</b>	<b>43</b>
3.A	Commandes internes, externes et le PATH	43
3.A.1	Commandes externes et PATH	43
3.A.2	Les commandes internes	44
3.A.3	Traitement du premier mot de la ligne de commandes	45
3.B	Identifier les commandes	46
3.B.1	type : obtenir la nature d'une commande	46
3.B.2	which : afficher le chemin complet d'une commande externe	47
3.B.3	whereis : rechercher les fichiers relatifs à une commande externe	48
3.C	Aide en ligne	49
3.C.1	help : aide sur les commandes internes de bash	49
3.C.2	man : le manuel en ligne	50
3.C.3	whatis/apropos : rechercher des pages de manuel	52
<b>4</b>	<b>Bash : composition d'une ligne de commandes</b>	<b>53</b>
4.A	Introduction	53
4.B	Les caractères spéciaux de bash	54
4.C	Expansion du tilde	55
4.D	Expansion des chemins de fichiers	56
4.E	Expansion des accolades	58
4.F	Protection par chaînes	59
4.F.1	Protection par quotes	59
4.F.2	Protection par guillemets	60
<b>5</b>	<b>Gestion des entrées-sorties</b>	<b>61</b>
5.A	Entrées-sorties par défaut	61
5.B	Affichage de texte et de fichiers	62
5.B.1	echo : afficher un message	62
5.B.2	cat : concaténer des fichiers	64
5.B.3	less : afficher le contenu de fichiers en paginant	66
5.C	Redirections	67
5.C.1	Redirection de l'entrée standard	67
5.C.2	Redirection de la sortie standard	68
5.C.3	Redirection de la sortie d'erreur	69
5.C.4	Redirection des sorties ensemble	70
5.D	tty : connaître son terminal	71



<b>6</b>	<b>Tubes et filtres</b>	<b>73</b>
6.A	Tubes (pipes)	73
6.B	Filtres	74
6.B.1	wc : compter les lignes, mots et octets de fichiers	75
6.B.2	head : écrire le début de fichiers	77
6.B.3	tail : écrire la fin de fichiers	78
6.B.4	cut : écrire des parties de lignes	80
6.B.5	sort : trier des lignes de fichiers	83
6.B.6	uniq : éliminer des lignes successives identiques	87
<b>7</b>	<b>Impressions sous Unix</b>	<b>91</b>
7.A	lpr : lancer une impression	91
7.B	lpstat : obtenir des informations sur les imprimantes	92
7.C	lpq : obtenir l'état d'une file d'attente d'impression	93
7.D	lprm : supprimer des impressions en attente	93
<b>8</b>	<b>Courrier électronique</b>	<b>95</b>
8.A	mail : la messagerie électronique traditionnelle d'Unix	95
8.B	Réexpédition des messages	96
8.C	Consulter une boîte aux lettres électronique	96
8.C.1	Obtenir de l'aide	98
8.C.2	Se déplacer dans la liste des en-têtes	98
8.C.3	Lire un message	98
8.C.4	Répondre à un message	98
8.C.5	Sauvegarder un message	98
8.C.6	Supprimer un message, annuler la suppression	99
8.C.7	Archiver les messages	99
8.C.8	Quitter sans modifier la boîte aux lettres	99
8.D	Envoyer un mail	99
8.D.1	Carbon copies	100
8.D.2	Blind Carbon copies	100
8.E	mail, une évolution de mail	100
<b>9</b>	<b>Personnalisation de l'environnement bash</b>	<b>101</b>
9.A	Les alias	101
9.A.1	alias : créer un alias d'une ligne de commandes	101
9.A.2	Empêcher l'expansion d'un alias	103
9.A.3	unalias : supprimer un alias	103
9.B	Introduction aux fonctions shell	104
9.C	La variable d'environnement PATH	105
9.D	Fichiers de personnalisation de bash	107
9.D.1	Fichiers exécutés au démarrage de bash	107
9.D.2	Rôle des fichiers de démarrage	108
9.D.2.a	Le fichier /etc/profile	108
9.D.2.b	Le fichier ~/.bash_profile	109
9.D.2.c	Extension des fichiers /etc/profile et ~/.bash_profile	109
9.D.2.d	Le fichier /etc/bash.bashrc	110
9.D.2.e	Le fichier ~/.bashrc	110
9.D.3	Comment rendre permanente la personnalisation de son environnement ?	110
9.D.3.a	Où placer les modifications de la variable PATH ?	110
9.D.3.b	Où placer la création des alias et des fonctions ?	111

9.D.3.c	Où placer la définition du masque des permissions avec umask ?	111
9.D.4	Particularité de l'environnement graphique	111
9.D.5	Fichier exécuté à l'arrêt d'un login shell	111
<b>10</b>	<b>Les processus</b>	<b>113</b>
10.A	Introduction	113
10.B	Vie d'un processus	113
10.B.1	Lancement par le shell d'un processus en premier plan	114
10.B.2	Lancement par le shell d'un processus en tâche de fond	114
10.C	État des processus	116
10.C.1	jobs : lister les processus lancés par le shell	116
10.C.2	ps : obtenir une liste de processus existants	116
10.C.3	lsuf : lister les fichiers ouverts	120
10.D	Gestion des processus	128
10.D.1	Agir sur un processus avec les caractères de contrôle	128
10.D.2	kill : envoyer un signal à un processus	128
10.D.3	pkill : envoyer un signal à des processus sélectionnés	131
10.D.4	killall : envoyer un signal aux processus exécutant certaines commandes	131
<b>11</b>	<b>Utilisateurs et groupes</b>	<b>133</b>
11.A	Gestion des comptes utilisateurs	133
11.A.1	Les fichiers /etc/passwd et /etc/shadow	133
11.A.2	useradd : créer un utilisateur	134
11.A.3	Modification du mot de passe utilisateur	134
11.A.3.a	passwd : changer son mot de passe local	134
11.A.3.b	smbpasswd : changer son mot de passe avec SAMBA	135
11.B	Gestion des groupes	136
11.B.1	Les fichiers /etc/group et /etc/gshadow	136
11.B.2	groupadd : créer un groupe	136
11.C	Gestion des identités	137
11.C.1	Identités réelle et effective	137
11.C.1.a	id : connaître son identité effective et réelle	137
11.C.1.b	whoami : connaître son nom d'utilisateur effectif	138
11.C.1.c	groups : connaître les groupes d'un utilisateur	139
11.C.2	su : utiliser une autre identité pour exécuter un shell ou une commande	139
11.D	Travailler avec les groupes	141
11.D.1	gpasswd : administrer un groupe	141
11.D.2	newgrp : nouveau shell en changeant de groupe par défaut	141
11.E	Informations sur les utilisateurs	142
11.E.1	who : savoir quels sont les utilisateurs connectés	142
11.E.2	finger : obtenir des renseignements sur des utilisateurs	144
11.E.3	w : savoir ce que font les utilisateurs sur leurs terminaux	145
11.E.4	last : connaître les connexions et déconnexions des utilisateurs	148
<b>12</b>	<b>Expressions régulières et utilitaires</b>	<b>151</b>
12.A	Expressions régulières	151
12.B	Utilitaires majeurs traitant les expressions régulières	153
12.B.1	grep : filtrer les lignes de fichiers	153
12.B.2	sed : éditer un flux de texte	157
12.B.3	awk : programmer le traitement de fichiers	163

<b>13 Autres manipulations de fichiers</b>	<b>173</b>
13.A Opérations diverses	173
13.A.1 file : analyser le contenu d'un fichier	173
13.A.2 split : découper un fichier en plusieurs fichiers	174
13.A.3 curl : télécharger ou télédéposer une URL	176
13.B Recherche de fichiers	178
13.B.1 find : rechercher des fichiers	178
13.B.2 locate : localiser rapidement un fichier par son nom	182
13.B.3 updatedb : créer ou mettre à jour une base pour locate	184
13.C Compression	184
13.C.1 compress/uncompress : la compression historique mais obsolète d'Unix	184
13.C.1.a compress : compresser un fichier	184
13.C.1.b uncompress : décompresser un fichier d'extension .Z	185
13.C.2 gzip/gunzip : la compression la plus répandue	185
13.C.2.a gzip : compression avec somme de contrôle	185
13.C.2.b gunzip : effectuer différentes décompressions	186
13.C.2.c Les commandes en « z »	186
13.C.3 bzip2/bunzip2 : la compression la plus récente	187
13.C.3.a bzip2 : plus d'efficacité et de robustesse dans la compression	187
13.C.3.b bunzip2 : décompresser les fichiers bzip2	187
13.C.3.c Les commandes en « bz »	188
13.D Archivage	188
13.D.1 tar : l'archiviste historique et incontournable d'Unix	188
13.D.2 zip/unzip : un archiviste récent et compatible WinZIP	190
<b>14 Compléments sur les fichiers</b>	<b>191</b>
14.A ln : créer un lien	191
14.A.1 Lien physique	191
14.A.2 Lien symbolique	195
14.B Compléments sur les permissions	197
14.B.1 Changer d'identité en exécutant un programme	197
14.B.1.a Le bit set-uid sur les fichiers	197
14.B.1.b Le bit set-gid sur les fichiers	200
14.B.1.c Combinaison des bits set-uid et set-gid sur les fichiers	201
14.B.2 Permissions supplémentaires sur les répertoires	201
14.B.2.a Le bit set-gid sur les répertoires	201
14.B.2.b Le bit set-uid sur les répertoires	202
14.B.2.c Le sticky-bit	202
14.B.3 chmod pour modifier les bits set-uid, set-gid et sticky	203
14.B.3.a Mode absolu	203
14.B.3.b Mode symbolique	205
<b>15 Bash : variables et tableaux</b>	<b>207</b>
15.A Les variables de type chaîne	207
15.A.1 Création d'une variable de type chaîne	207
15.A.2 unset : supprimer une variable ou fonction	210
15.A.3 Substitution des variables	211
15.A.4 Substitutions étendues des variables	213
15.A.4.a Substitutions selon l'existence et le contenu	213
15.A.4.b Substitutions de la longueur et d'une sous-chaîne	215

15.A.4.c Substitutions de sous-chaînes avec les motifs . . . . .	216
15.B Les tableaux . . . . .	217
15.B.1 Création d'un tableau . . . . .	218
15.B.2 Suppression d'un tableau ou d'un élément . . . . .	219
15.B.3 Substitutions propres aux tableaux . . . . .	219
15.B.3.a Obtenir tous les éléments d'un tableau . . . . .	219
15.B.3.b Obtenir la liste des indices des éléments d'un tableau . . . . .	221
15.B.3.c Obtenir le nombre d'éléments d'un tableau . . . . .	221
15.B.3.d Application des substitutions de sous-chaînes à l'ensemble d'un tableau . . . . .	222
15.C Afficher les variables, tableaux et fonctions existants . . . . .	223
15.D Variables et tableaux internes de bash . . . . .	224
<b>16 Les scripts et la programmation bash . . . . .</b>	<b>227</b>
16.A Les scripts bash . . . . .	227
16.A.1 Exécution d'un script . . . . .	227
16.A.2 La ligne shebang . . . . .	229
16.A.3 Les commentaires . . . . .	229
16.A.4 Paramètres positionnels et paramètres spéciaux . . . . .	230
16.A.5 Substitutions de sous-chaînes et paramètres \$@ et \$* . . . . .	232
16.A.6 Paramètres et variables dans les scripts et fonctions . . . . .	233
16.A.6.a Paramètres dans les scripts et fonctions . . . . .	233
16.A.6.b Portée des variables et tableaux dans les scripts et fonctions . . . . .	234
16.A.6.c local : créer des variables et des tableaux locaux . . . . .	235
16.B Valeur de retour et suites de commandes . . . . .	237
16.B.1 Valeur de retour des commandes . . . . .	237
16.B.1.a return : indiquer une valeur de retour pour une fonction . . . . .	238
16.B.1.b exit : indiquer une valeur de retour pour un script . . . . .	239
16.B.1.c Valeur de retour d'un pipeline . . . . .	239
16.B.2 Négation de la valeur de retour d'une commande ou d'un pipeline . . . . .	239
16.B.3 Suites de commandes . . . . .	240
16.C Tester des conditions . . . . .	242
16.C.1 Tester des conditions arithmétiques . . . . .	242
16.C.2 test : tester des conditions de différentes natures . . . . .	243
16.C.3 Les commandes true et false . . . . .	247
16.D Contrôle de flux pour la programmation bash . . . . .	247
16.D.1 L'instruction if . . . . .	247
16.D.2 Les boucles for . . . . .	249
16.D.2.a La boucle for de sh . . . . .	249
16.D.2.b La boucle for des langages évolués . . . . .	252
16.D.3 La boucle while . . . . .	253
16.D.4 La boucle until . . . . .	254
16.D.5 Instructions de court-circuitage du déroulement d'une boucle . . . . .	254
16.D.5.a L'instruction break pour sortir d'une boucle . . . . .	254
16.D.5.b L'instruction continue pour passer à l'itération suivante . . . . .	255
16.D.6 L'instruction case pour les branchements multiples . . . . .	256
16.D.7 L'instruction select pour créer des menus . . . . .	257
16.E Gestion des entrées/sorties d'un script . . . . .	260
16.E.1 Écriture de messages d'erreur . . . . .	260
16.E.2 read pour lire des entrées . . . . .	260
16.E.3 Redirection des entrées/sorties d'une instruction composée ou d'un bloc . . . . .	262

16.E.4 Les fichiers virtuels /dev/stdin, /dev/stdout et /dev/stderr	265
16.F Traitement des options d'un script	266
16.F.1 L'instruction shift pour décaler les arguments	266
16.F.2 Exemple de script : ndistargs	268
<b>17 Environnement des processus</b>	<b>271</b>
17.A Définition	271
17.B Héritage	271
17.C Les pipes et l'environnement des processus	272
17.D Utilité des variables, tableaux et fonctions d'environnement	273
17.E Création de variables, tableaux et fonctions d'environnement	273
<b>18 Autres fonctionnalités de bash</b>	<b>275</b>
18.A Création d'un sous-shell en utilisant les parenthèses	275
18.B Substitution de commande	276
18.C Substitution d'expressions arithmétiques	278
18.D Décomposition en mots	278
18.E Traitement de la ligne de commandes	280
<b>A L'éditeur vi</b>	<b>281</b>
A.1 Fonctionnement	281
A.2 Entrée dans le mode insertion	282
A.3 Entrée dans le mode remplacement	282
A.4 Sortie du mode insertion et du mode remplacement	282
A.5 Mode commandes	282
A.5.1 Commandes édition de fichier	283
A.5.2 Déplacements	283
A.5.3 Effacements	285
A.5.4 Remplacement	285
A.5.5 Copier-coller	285
A.5.6 Substitution de chaînes	285
A.5.7 Commande globale	286
A.5.8 Partage de la fenêtre vi	286
A.5.8.a Création de sous-fenêtres	286
A.5.8.b Fermeture d'une (sous-)fenêtre	287
A.5.8.c Déplacement entre sous-fenêtres	287
A.5.9 Divers	287
A.5.9.a Aide de l'éditeur	287
A.5.9.b Coloration syntaxique	287
A.5.9.c Options de l'éditeur	288
A.5.9.d Fichier de configuration	288
A.5.10 Facilités pour la programmation	289
A.5.10.a Complétion automatique	289
A.5.10.b Recherche d'un identificateur	289
A.5.10.c Déplacement jusqu'à un identificateur et retour à la position initiale	289
A.5.10.d Indentation	289
A.6 Mode visuel	289

<b>B</b>	<b>Gestion des patches</b>	<b>291</b>
B.1	Introduction . . . . .	291
B.2	diff : comparer des fichiers . . . . .	291
B.3	patch : appliquer ou annuler un patch . . . . .	296
	<b>Index</b>	<b>301</b>

# Chapitre 1

## Prise en mains

---

### 1.A Introduction

Notre cadre d'étude sera le système GNU/Linux, et en particulier la distribution Debian Squeeze/Sid. L'installation du système n'est pas l'objet de ce cours. L'utilisation de l'environnement graphique (notamment le bureau) non plus, bien qu'il puisse être fait allusion, de-ci, de-là, à des applications graphiques.

Ce manuel concerne principalement certaines notions clés des systèmes Unix, telles que les utilisateurs, les fichiers, les processus, les entrées-sorties, les tubes, les interpréteurs de commandes, mais aussi un nombre important de commandes utilisées régulièrement par les utilisateurs et les administrateurs d'un système Unix.

Par le terme commande, on désigne principalement ici un programme sans interface graphique, que l'on exécute et que l'on paramètre depuis un interpréteur de commandes en mode texte. Pour certaines commandes, il existe des applications présentant une interface graphique permettant de les utiliser. Cependant, ces interfaces ne permettent que très rarement de régler aussi finement leurs paramètres qu'on ne peut le faire depuis un interpréteur de commandes en mode texte. De plus, nous verrons que ce type d'interpréteur permet de combiner des commandes afin de réaliser des tâches complexes et inédites. Cela repose sur la philosophie Unix selon laquelle un outil doit être spécialisé dans une tâche mais doit faciliter son insertion dans une chaîne de traitement (tube) composée d'autres outils.

### 1.B Notion de système d'exploitation

Un **système d'exploitation** (*Operating System*) est un logiciel qui est chargé en mémoire lors du démarrage d'un ordinateur et qui a pour but de gérer l'ensemble des ressources de ce dernier afin de les mettre à disposition d'un ou de plusieurs utilisateurs. Par ressources, on entend le processeur (appelé CPU), la mémoire, les disques et les périphériques (clavier, souris, moniteur, imprimantes, scanners, réseau, lecteurs/graveurs de CD/DVD/disquettes, stockage externe, etc.).

Au début de l'informatique, les ordinateurs étaient utilisés sans système d'exploitation. Sur un ordinateur, les programmes étaient exécutés les uns après les autres, chacun devant intégrer le code nécessaire à la gestion des ressources de l'ordinateur, ce qui les alourdissait considérablement. Il est rapidement devenu évident qu'il était plus efficace d'isoler la gestion matérielle pour la rendre résidente en mémoire et disponible aux programmes qui s'exécutent. De là sont apparus les systèmes d'exploitation.

Un système d'exploitation est dédié à une architecture matérielle. Il intègre les pilotes (*drivers*) des péri-

phériques qu'elle admet, et qui sont nécessaires à leur utilisation. Il organise la mémoire vive (RAM) pour la mettre à disposition des programmes selon leurs besoins. Il structure les disques durs et les unités de stockage afin d'y sauvegarder l'information sous la forme de fichiers. Il fournit une interface de programmation (*Application Programming Interface* ou API) permettant d'écrire des programmes en s'appuyant sur les services fournis par le système, comme par exemple créer un fichier ou communiquer à travers un réseau. Il permet de charger en mémoire des programmes présents sur disque (ou autre périphérique de stockage) et de les exécuter sous la forme de processus. Il orchestre l'exploitation des ressources par les processus.

Parmi les caractéristiques que les systèmes peuvent avoir on peut noter :

- multi-tâches : un système est multi-tâches s'il peut exécuter plusieurs programmes simultanément, même en présence d'un seul processeur. Le processeur n'est alloué à l'exécution d'un processus que pour une petite durée, passée laquelle le processus est placé en attente, et le processeur alloué à l'exécution d'un autre processus. L'ordonnanceur (*scheduler*) du système orchestre l'exploitation du processeur en gérant une file d'attente des processus. Les systèmes d'exploitation modernes sont multi-tâches ;
- multi-utilisateurs : un système est multi-utilisateurs s'il permet à plusieurs utilisateurs d'exécuter des programmes simultanément. Dans ce cas, le système doit être non seulement multi-tâches mais doit aussi garantir un minimum de sécurité, afin qu'un utilisateur ne puisse nuire aux autres.

Les architectures matérielles ont beaucoup évolué et les systèmes aussi. De nos jours, la plupart des systèmes sont multi-tâches, multi-utilisateurs, et peuvent gérer plusieurs processeurs. Ceux des ordinateurs personnels (Windows, Mac OS, GNU/Linux) offrent en outre une interface graphique ergonomique, gèrent le son, la vidéo (2D ou 3D) et l'accès à des réseaux tels qu'Internet.

Enfin, les techniques récentes de virtualisation offrent désormais la possibilité à un système dit hôte, d'héberger (d'exécuter) un ou plusieurs autres systèmes dits invités. Du point de vue des administrateurs système/réseaux, la virtualisation offre de nombreux avantages en termes de sécurité et de maintenance, et est de plus en plus mise en œuvre.

## 1.C Unix et GNU/Linux

Le système Unix a été initialement conçu dans les laboratoires Bell dans les années 70 comme un système multi-tâches et multi-utilisateurs. La version d'Unix principalement étudiée dans ce document est **GNU/Linux**, qui est un clone d'Unix. Linux a été développé en 1991 par un étudiant finlandais du nom de Linus Torvalds qui s'est inspiré du système d'exploitation Minix développé par le Professeur Andrew Tanenbaum, pour écrire un Unix fonctionnant sur une architecture Intel 80386. Minix était aussi un clone d'Unix mais n'était pas vraiment utilisable car il a été développé à des fins principalement pédagogiques.

En revanche, Linus voulait que Linux soit un jour assez mûr pour pouvoir être utilisé par le plus grand nombre, et cela gratuitement. En 1991, il s'est mis à diffuser les sources de son système à quiconque en voulait une copie de façon à ce que d'autres développeurs se joignent à lui pour l'améliorer. Depuis, Linux n'a cessé d'évoluer et a bénéficié du soutien de grandes entreprises de l'informatique qui contribuent à son développement. Il bénéficie d'une très bonne réputation de stabilité et de performances depuis quelques années déjà. Le nombre de ses utilisateurs (entreprises et particuliers) ne cessant de croître, certains fabricants de matériels et éditeurs de logiciels commercialisent des versions Linux de leurs produits. Ces produits sont encore rares en comparaison des offres pour Windows et Mac OS, mais se font de plus en plus nombreux.

Ses sources étant librement disponibles, Linux a été "porté" sur de nombreux autres architectures que les PC.



C'est pourquoi, on le retrouve beaucoup dans l'informatique embarquée <sup>1</sup>.

Linux en lui-même n'est pas le système d'exploitation : il en est le cœur, le noyau (*kernel* en anglais). Le système est en réalité GNU/Linux. GNU (qui veut dire « *GNU's Not Unix* », un acronyme récursif) est un projet lancé en 1984 et soutenu par la *Free Software Foundation* pour développer un système libre de type Unix. Le noyau GNU appelé Hurd n'est pas encore totalement arrivé à maturité. En revanche, le projet GNU a conduit au développement de très nombreux outils logiciels libres pour les systèmes Unix et assimilés (y compris Linux et Hurd). Ces outils servent d'interface entre le kernel et les utilisateurs, et permettent de réaliser des tâches très diverses comme copier des fichiers, éditer des fichiers, compiler des programmes, gérer le système, etc.

Par la suite nous parlerons de Linux pour désigner GNU/Linux. Linux est distribué par de nombreux éditeurs : Red Hat, Mandriva, Debian, SuSE, qui ont chacun apporté leur touche au système. Principalement, c'est une adaptation du kernel, l'organisation (systèmes de fichiers, fichiers de configuration, ...) et l'"enrobage" du système qui change d'une distribution à l'autre, notamment le bureau, les outils d'administration du système, d'impression, de navigation, etc.

Toutes ces distributions ne sont toutefois que des déclinaisons du système Unix avec lequel il est difficile de faire des comparaisons. Ainsi, une grande partie de ce cours est valable pour toutes les versions d'Unix.

## 1.D Utilisateurs et groupes Unix

Plusieurs utilisateurs peuvent utiliser en même temps un ordinateur sous Unix, et y exécuter en parallèle des programmes. Le système gère l'exploitation de la CPU et de la mémoire (et du matériel en général), pour que les utilisateurs se les partagent sans se gêner les uns les autres. À part la charge du système (sa réactivité), l'activité de multiples utilisateurs est transparente. Des mécanismes de sécurité assurent la confidentialité des données et la protection des activités : un utilisateur ne peut *a priori* pas nuire à l'activité d'autrui. Ces mécanismes reposent sur la notion d'utilisateur et de groupe, et des droits qui leur sont accordés.

Sous Unix, tout utilisateur est identifié de manière unique par son **numéro d'utilisateur**, appelé **UID** (*user identifier*), auquel est associé un nom d'utilisateur <sup>2</sup>, appelé aussi nom de *login*. Il appartient toujours à au moins un groupe, lui aussi identifié par un **numéro (GID)** et un **nom de groupe**.

Le principe est que le numéro et les groupes d'un utilisateur déterminent ses possibilités d'action : accès à certaines "applications" et pas à d'autres, accès à des fichiers, périphériques, etc. L'utilisateur est toujours maître de ses propres activités, mais ne peut intervenir sur celles des autres.

Il existe un utilisateur particulier appelé **root** (dont le numéro est 0) qui appartient au groupe **root** (numéro 0). C'est le **superviseur du système**. Il est privilégié car il possède potentiellement tous les droits sur le système et sur les utilisateurs. Il peut intervenir sur toutes les activités du système. Il peut créer, modifier ou supprimer à volonté des groupes (commandes **groupadd**, **groupmod** et **groupdel**) et des utilisateurs (commandes **useradd**, **usermod** et **userdel**).

Lors de la création d'un utilisateur, **root** indique (généralement) son nom et son groupe, son nom réel, ainsi qu'un mot de passe, un répertoire d'accueil et un interpréteur de commandes :

- le **mot de passe** sert à l'authentification d'un utilisateur lorsque celui-ci se loge (accède au système) ;

---

1. système numérique aux ressources limitées, destiné à une tâche précise (lecteur multimédia, Box ADSL, téléphone, navigateur GPS, ordinateur de bord, etc).

2. Son nom réel (civil) n'a aucune importance pour le système. Sa connaissance relève de la responsabilité de l'administrateur du système. À l'avenir, nous nous limiterons à écrire "nom" pour désigner le "nom d'utilisateur".

- le **répertoire**<sup>3</sup> **d'accueil** appelé aussi **répertoire personnel** (*home directory*) ou encore **répertoire de travail initial**, est un espace réservé à l'utilisateur, dans lequel il peut stocker ses informations personnelles. Nous y reviendrons un peu plus tard ;
- l'**interpréteur de commandes** est un programme permettant d'exécuter des commandes complexes. Comme il en existe plusieurs sous Unix, cela permet de spécifier l'interpréteur que veut utiliser l'utilisateur.

## 1.E Pour les utilisateurs de Microsoft Windows

Les utilisateurs de Windows auront sûrement fait le rapprochement entre l'utilisateur root de Linux et l'Administrateur de Windows, et leur rôle est effectivement le même.

Ces comptes "privilegiés" ne doivent être utilisés que lorsque nécessaire, notamment pour installer des applications ou des périphériques, gérer le système et les autres utilisateurs. Normalement, aucune action pouvant mettre en péril la sécurité du système ne doit être initiée en tant qu'administrateur. Cela comprend notamment le surf sur Internet, mais pas seulement.

Si cette politique est généralement suivie par les particuliers ayant installé un Unix chez eux, elle ne l'est pratiquement jamais par ceux qui utilisent Windows. En effet, sur Windows (peut être pas Seven que je n'ai pas eu l'occasion de tester), un utilisateur est créé par défaut en tant qu'administrateur. Ce qui veut dire que selon ses actions (comme l'ouverture d'une pièce jointe avec un Troyen, un Virus ou autre joyeuseté), il compromettra le système entier et les données des autres utilisateurs. Il est toutefois possible de restreindre les droits d'un utilisateur mais cela nécessite de le faire explicitement, ce que l'utilisateur lambda ne fait généralement pas, notamment car il ne le sait pas.

La situation est d'autant plus délicate que certains éditeurs de logiciels ne se préoccupent pas beaucoup de la notion d'utilisateurs et il n'est pas rare d'installer une application qui, sans véritable raison, ne peut être exécutée qu'avec un compte administrateur. De ce fait, utiliser un compte sans privilège peut s'avérer fastidieux sous Windows, mais la sécurité du système passe par là, ainsi que le recommande Microsoft...

Le compte utilisateur dont vous disposez sous Linux et Windows au Département est sans privilège. Vous pourrez constater que cela est largement suffisant pour vos tâches quotidiennes.

## 1.F Interpréteur de commandes

Un **interpréteur de commandes sert d'interface entre le système et l'utilisateur**. C'est un environnement qui permet à l'utilisateur de demander au système de réaliser certaines tâches. L'environnement graphique peut être considéré comme un interpréteur de commandes **événementiel** car il déclenche des actions selon des clics ou des déplacements de souris. Cependant, les commandes réalisées de cette manière sont basiques : exécution d'un programme, déplacement dans un répertoire, ouverture d'un fichier...

Les **interpréteurs de commandes en mode texte** sont des programmes qui lisent les ordres formulés par l'utilisateur au moyen du clavier et les appliquent. Avant l'apparition des modes graphiques sur les écrans, les systèmes ne pouvaient être utilisés que par le biais de ce type d'interpréteur. Après son démarrage, le système attendait que l'utilisateur tape des ordres au clavier, par exemple pour exécuter un programme. Le premier système développé pour les PC a été **MS-DOS** qui fournissait un interpréteur de commandes très basique. L'application

---

3. Un répertoire est aussi appelé un **dossier**.

graphique **cmd.exe** des systèmes Windows exécute un interpréteur similaire dans une fenêtre.

Les interpréteurs de commandes d'Unix sont réputés pour permettre l'expression d'ordres complexes conduisant à des traitements évolués, qu'on ne peut déclencher aussi aisément avec un environnement graphique. Cette puissance expressive a été l'un des moteurs du succès des systèmes Unix. On appelle ces interpréteurs des **shells** (coquilles en français). En fait, le **shell** (sh) est, avec le **C-shell** (csh), le premier interpréteur de commandes évolué d'Unix. Puis vinrent le Korn-shell (ksh), tcsh, bash... Par abus de langage, on dit souvent «un shell est lancé» lorsqu'on veut dire «un interpréteur de commandes est exécuté».

L'interpréteur de commandes étudié dans ce document s'appelle **bash** (pour *Bourne-Again SHell*). Il est disponible sur tous les systèmes Unix récents. Bash est une version évoluée de **sh** dont il inclut toutes les caractéristiques. Il offre en outre certaines facilités intéressantes comme l'utilisation des flèches pour rappeler d'anciennes commandes. Une bonne partie de ce cours reste valable pour **sh**.

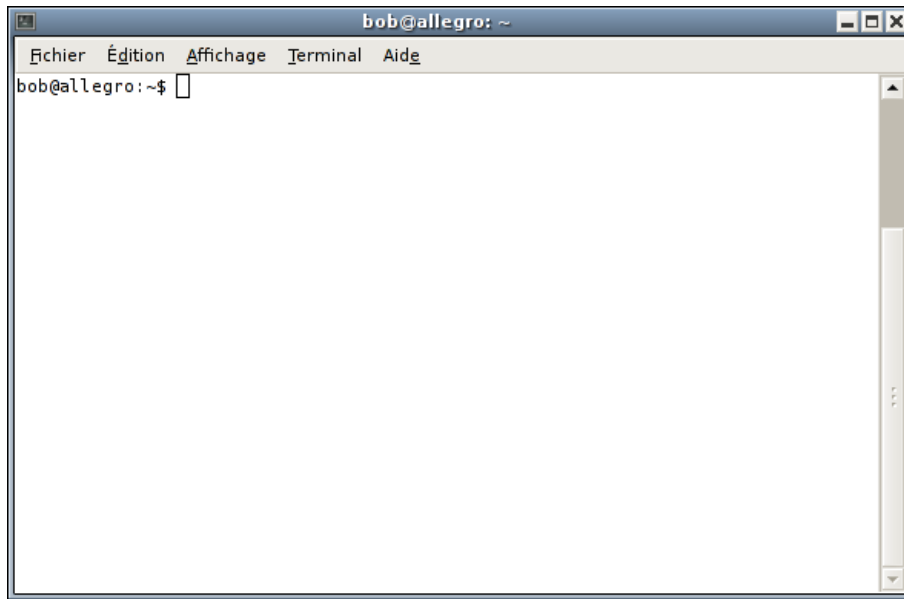
## 1.F.1 Terminal

L'interaction avec le shell se fait par l'intermédiaire d'un **terminal**. Historiquement, un terminal est un équipement informatique équipé d'un clavier et d'un écran. Différents terminaux ont été commercialisés. Le succès du terminal VT100 de DEC en a fait un standard de fait. Il possède un écran de 80 colonnes et 24 lignes (pouvant basculer en 132 colonnes et 14 lignes), et reconnaît des séquences spéciales de caractères ASCII permettant notamment de modifier des attribus vidéos (gras, souligné, vidéo inverse, clignotant), d'effacer l'écran et de positionner le curseur.



Bien souvent le système Linux finit son démarrage en simulant la présence de 6 terminaux texte de type VT100 et un terminal graphique. Les terminaux texte sont accessibles par les combinaisons de touche **CRTL+ALT+F1** à **CRTL+ALT+F6**. Ils ne permettent pas l'emploi de la souris. Le premier terminal, appelé **tty1**, affiche les (derniers) messages du démarrage.

Le terminal graphique est celui affiché après le démarrage quand apparaît la fenêtre invitant l'utilisateur à s'authentifier. Il est accessible par la combinaison **CRTL+ALT+F7**. Une fois l'utilisateur authentifié, ce terminal affiche un bureau graphique. Depuis ce bureau, il est possible d'exécuter une application graphique qui ouvre une fenêtre simulant un terminal physique de type VT100 :



❗ Il existe de nombreuses applications graphiques de type *terminal* : **xterm**, **rxvt**, **aterm**, **gnome-terminal**, **konsole**, .... Pour le bureau **Gnome**, l'application par défaut est **gnome-terminal**. Elle est accessible depuis le menu *Applications* → *Accessoires* → *Terminal*.

## 1.F.2 Invite de commandes

Lorsqu'un interpréteur de commandes est exécuté, il invite l'utilisateur à entrer une **ligne de commande** en affichant une **invite de commande** (ou *prompt*, en anglais). Bash affiche généralement une séquence de caractères ayant la forme suivante :

*nom@hôte : rép\$*

où :

- *nom* est le nom de l'utilisateur ayant lancé bash ;
- *hôte* est le nom de la machine sur laquelle bash est exécuté ;
- *rép* est le répertoire de travail. Au départ, c'est le répertoire d'accueil de l'utilisateur. Ensuite, il change selon les commandes tapées. Nous y reviendrons plus tard.

Ainsi, en supposant que le répertoire d'accueil de l'utilisateur bob sur allegro est `/home/bob`, alors le prompt pour bob sur allegro devrait ressembler au début à :

**bob@allegro: /home/bob\$**

Cependant, pour sh et bash (mais aussi pour d'autres interpréteurs) le répertoire d'accueil de l'utilisateur en session (celui qui a lancé bash) peut être désigné sous certaines conditions par le **caractère spécial tilde** `~` (voir section 4.B page 54). Ainsi, au début, le prompt ressemblerait plutôt à ceci :

**bob@allegro: ~\$**

On peut alors taper une ligne de commandes dans la fenêtre concernée. Celle-ci sera ensuite analysée par bash et si elle est correcte, elle sera exécutée après avoir été éventuellement transformée<sup>4</sup>. Lorsque l'exécution de la

4. Ces transformations, dues aux caractères spéciaux, sont étudiées tout au long de ce document.

commande est terminée, bash affiche à nouveau le prompt pour signifier qu'il est prêt à interpréter une nouvelle ligne de commandes.

### Exemple

```
bob@allegro:~$ echo hello world
hello world
bob@allegro:~$
```

➡ après avoir exécuté la commande **echo hello world**, qui affiche «hello world», le shell affiche à nouveau le prompt.

□

❗ Le prompt n'est pas figé. Chacun peut le personnaliser comme il le souhaite (on verra comment lorsque nous verrons la variable **PS1**). Notamment, par convention le dernier caractère du prompt de l'utilisateur root (administrateur) est **#** (plutôt que **\$**). Afin de ne pas surcharger les exemples du poly, le prompt sera souvent réduit à un simple **\$**, à moins que son écriture exhaustive apporte une information d'intérêt.

💣 Dans le document, les exemples faisant intervenir des commandes nécessitant les droits de root sont présentés avec un prompt se terminant par **#**.

## 1.F.3 Ligne de commandes

C'est une suite de caractères terminée par l'appui de la touche **Entrée**. Elle peut définir une commande ou une composition de commandes (commande complexe).

Pour l'instant, nous nous limiterons à l'expression de commandes simples. Mais avant d'aller plus loin, voyons les conventions d'écriture que nous utiliserons.

### 1.F.3.a Convention de description d'une commande

Dans ce document, de nombreuses commandes Unix seront expliquées et une forme générale de leur utilisation (appelée **synopsis**) sera aussi indiquée. Ces formes suivent une **grammaire** conforme aux conventions suivantes :

- **expression** : une expression écrite en **gras** est terminale, c'est à dire qu'elle doit apparaître telle quelle. Les caractères écrits en gras (**[**, **]**, ...) rentrent dans cette catégorie ;
- *expression* : une expression écrite en italique est non-terminale, c'est à dire qu'elle doit être remplacée par une suite de caractères. Une grammaire définissant ce qu'*expression* peut valoir est mentionnée (lorsque nécessaire) ;
- [*expression*] : une *expression* entre crochets (non gras !) est optionnelle ; elle peut apparaître 0 ou 1 fois (c'est à dire : au plus une fois) ;
- {*expression*} : les accolades (non grasses !) indiquent que l'on peut écrire 0 ou plusieurs fois quelque chose correspondant à *expression* mais en les séparant par des **blancs** ;
- *expression*<sub>1</sub> | *expression*<sub>2</sub> : la barre droite (non grasse) indique une alternative entre les deux *expressions*. On doit utiliser *expression*<sub>1</sub> ou *expression*<sub>2</sub>, mais pas les deux ;

- (*expression*) : les parenthèses (non grasses) permettent de grouper des *expressions*. Elles seront principalement utilisées pour lever les ambiguïtés, notamment en présence de | ;
- d'une façon générale, les expressions (ou caractères) non écrits en gras ne font pas partie du langage. Elles sont présentes pour définir la grammaire. C'est notamment le cas des expressions non-terminales (en italiques).



Lorsque les expressions ne sont pas collées, c'est qu'il faut les séparer par un ou plusieurs séparateurs comme l'espace ou la tabulation, appelés des blancs. La première étape de l'interprétation d'une ligne de commandes par le shell est sa décomposition en éléments appelés jetons (*tokens*). Notons que parfois, il sera fait usage de la marque `_` pour représenter explicitement un espace.



Il s'agit ici d'une convention (notation) exploitant les polices de caractères (gras, italiques ou normales), dérivée de la notation BNF (Backus-Naur-Form).

### Exemple

La description ci-dessous, d'une commande (totalement imaginaire) qui s'appellerait **cmd** :

**cmd** [*-a*] [*-u*] [*-v*] [*-9*] {*argument*}

veut dire que pour utiliser **cmd**, il faut taper **cmd** suivi éventuellement de *-a*, suivi éventuellement de *-u*, suivi éventuellement de *-v*, suivi éventuellement de *-9*, suivi de 0 ou plusieurs arguments séparés. Ici, la forme (grammaire) d'un argument n'est pas décrite.

□

### 1.F.3.b Découpage d'une ligne de commande en nom, options et arguments

Une ligne de commande (simple) tapée sous Unix peut être découpée en trois parties : son nom, les options et les arguments.



Le nom est en tête de la (ligne de) commande, suivi des options puis des arguments.

### Exemple

Une utilisation probable de la commande **cmd** définie précédemment figure ci-dessous en gras, après le prompt :

bob@allegro:~\$ **cmd -a -v -9 arg1 arg2 arg3**

Après l'appui de la touche Entrée, le shell découpe la ligne de commande en **mots**, séparés (notamment) par des blancs (espace ou tabulation). La ligne précédente est donc composée de 7 mots. Le premier correspond à la commande que le shell doit exécuter. Il s'agit de **cmd**. En l'exécutant, le shell lui communique les 6 autres mots de la ligne de commande.



C'est à la commande (ici, **cmd**) de faire le tri parmi ces mots et de distinguer les options (a priori *-a*, *-v* et *-9*) des arguments (a priori *arg1*, *arg2* et *arg3*).

□

Tout ceci appelle les remarques suivantes :

- **le nom d'une commande** est soit une commande "connue" de bash ou le chemin complet d'un fichier exécutable. Nous reviendrons sur ces éléments plus tard ;
- **les options** modifient le comportement de la commande. Elles doivent être précédées du caractère tiret – sauf rares exceptions. Elles sont désignées par une lettre ou un chiffre. Par exemple, **-v** est l'option désignée par la lettre **v**, **-9** est l'option désignée par le chiffre **9**, **-v -9** est la composition de ces deux options. On peut aussi l'écrire **-v9** (c'est même parfois obligatoire, alors mieux vaut utiliser cette notation).

**i** Il est rare que l'ordre d'apparition des options soit imposé par la commande (comme c'est le cas dans l'exemple précédent). Cependant, pour certaines commandes, ce peut être important.

- **les arguments** sont les données (ou une partie) sur lesquelles la commande doit s'appliquer. Cela peut être plusieurs noms de fichiers à traiter par la commande (le plus souvent), du texte, rien du tout. . .



**La différence entre options et arguments est importante. Les options se composent alors que les arguments sont indépendants. La commande finale qui sera appliquée sur les arguments est la commande elle-même dont le comportement est modifié par la composition des options. Cette commande modifiée peut être alors appliquée sur les arguments, soit l'un après l'autre, soit dans leur globalité (plus rare).**

### Exemple

Reprenons la commande **echo** (vue plus en détail dans la section [5.B.1](#)). Nous avons vu qu'elle affiche du texte puis va à la ligne. Elle admet l'option **-n** lui demandant de ne pas aller à la ligne et l'option **-e** lui demandant de traiter spécialement certaines séquences de caractères :

```
$ echo 'hello\vworld'
```

```
hello\vworld
```

```
$
```

⇒ affiche le message et va à la ligne

```
$ echo -n 'hello\vworld'
```

```
hello\vworld$
```

⇒ affiche le message mais sans aller à la ligne (option **-n**) : le prompt est affiché à la suite sur la même ligne

```
$ echo -e 'hello\vworld'
```

```
hello
```

```
world
```

```
$
```

⇒ affiche le message en traitant spécialement (option **-e**) la séquence `\v`, provoquant une tabulation verticale après «hello», puis va à la ligne

```
$ echo -en 'hello\vworld'
```

```
hello
```

```
world$
```

⇒ composition des options **-e** et **-n** : le message est affiché en traitant spécialement certaines séquences mais sans aller à la ligne

□

### 1.F.3.c Raccourci de la description

Lorsqu'il n'y a pas d'ambiguïté, nous raccourcirons l'écriture des options dans la description des commandes. Par exemple, la description :

```
cmd [-a] [-v] [-(u|9)] {argument}
```

sera dorénavant écrite ainsi :

```
cmd [-av(u|9)] {argument}
```

Nous considérerons que cette écriture n'impose plus d'ordre sur l'apparition des options ainsi regroupées, sauf indication contraire.



# Chapitre 2

## Gérer les fichiers

### 2.A Les types de fichiers

Sous Unix, il existe 7 types de fichiers, regroupés en 3 catégories : les fichiers ordinaires, les répertoires et les fichiers spéciaux :

- les **fichiers ordinaires** (ou **fichiers réguliers**) sont destinés à contenir du texte, le code exécutable d'un programme, ou tout type d'information (données **binaires**, c'est à dire non texte). Unix ne fait pas de supposition sur leur contenu et ne tient pas compte de leur extension <sup>1</sup>. Même un fichier sans extension et contenant du texte peut éventuellement être exécuté ;
- les **répertoires** (ou **dossiers**) permettent d'organiser les fichiers du système. En effet, ce sont des fichiers dont le rôle est de contenir d'autres fichiers (spéciaux, ordinaires ou répertoires). Cela permet notamment de regrouper dans un même répertoire les fichiers ayant des caractéristiques communes (même propriétaire, les programmes, les documentations, ...), et de hiérarchiser les fichiers du système ;
- les **fichiers spéciaux** regroupent les 5 autres types de fichiers. Sans rentrer dans les détails, ce sont :
  - ◊ les **liens symboliques** (ou raccourcis) : fichiers désignant un autre fichier (voir section 14.A, page 191) ;
  - ◊ 2 types de fichiers spéciaux représentent les périphériques (auxquels sont associés des pilotes ou *drivers*) :
    - les **fichiers de périphériques en mode caractère** représentent notamment les claviers, souris, terminaux, modems, imprimantes, etc ;
    - les **fichiers de périphériques en mode bloc** représentent notamment les disques durs (et leurs partitions), clés USB, CD/DVD, etc ;

**i** Pour rester simple, en mode bloc, la lecture et l'écriture se font par blocs d'information (un nombre fixé d'octets) et à des emplacements précis. En mode caractère, on ne lit ou n'écrit que des octets, au fur et à mesure.

- ◊ 2 types de fichiers spéciaux servent à la communication entre les processus du système : les **fichiers socket** (ou socket unix) et les **tubes nommés** (appelés FIFO).

On constate que par principe, **sous Unix tout est fichier !** Cette particularité facilite l'écriture de programmes, où l'envoi d'une page à l'impression, ou d'un message à travers Internet se fait comme l'écriture dans un fichier ordinaire ; le noyau se chargeant du reste. Notons que les fichiers périphériques et de communication ont la particularité de pouvoir produire et/ou consommer de l'information.

1. Cela ne veut pas dire que l'extension d'un fichier n'a pas d'intérêt. Elle permet aux utilisateurs de rapidement connaître le contenu d'un fichier. Elle est aussi utilisée par les gestionnaires de fichiers graphiques de type **Nautilus** lorsqu'on double-clique sur un fichier.

## 2.B Organisation des fichiers

Les répertoires servent à organiser les fichiers du système et de créer une **hiérarchie arborescente**. La figure 2.1 présente un extrait de ce que pourrait être l'arborescence d'une machine, appelée *allegro*, dont on se servira régulièrement dans ce document. Les répertoires y figurent en gras. Le répertoire / (barre oblique appelée *slash*) en haut de la figure est appelé le **répertoire racine** ou **la racine**. Il est immuable et ne peut être placé dans un autre répertoire. On voit bien la structure dite arborescente.

 Sous Unix, il n'existe pas d'unité de disque (comme **C:** ou **D:** sous Windows). Les disques/partitions apparaissent comme des répertoires dans cette arborescence (voir section 2.D).

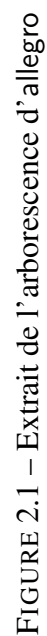
Tous les caractères sont autorisés pour former le nom d'un fichier, excepté '/' (*slash*). Le système est sensible à la casse : `toto` et `Toto` désignent deux fichiers différents<sup>2</sup>. Un fichier est **caché** si son nom commence par un point (comme `.bashrc`). Il reste accessible mais n'est "découvert" que si explicitement demandé.

Sur la figure, la racine contient 17 répertoires. On retrouve la plupart d'entre eux sur de nombreux Unix. Chacun a un rôle bien défini :

- **bin** contient un certain nombre de commandes (outils) élémentaires accessibles à tous les utilisateurs ;
- **boot** contient les fichiers nécessaires à l'amorçage (démarrage) du système ;
- **dev** contient les fichiers spéciaux de périphériques ;
- **etc** contient les fichiers de configuration du système ;
- **home** contient les répertoires personnels des utilisateurs ;
- **lib** contient les bibliothèques partagées<sup>3</sup> élémentaires et les modules du noyau ;
- **media** est le répertoire où apparaissent les médias amovibles (clés USB, CD/DVD, etc.) lorsqu'ils sont branchés ;
- **mnt** est le répertoire où l'on monte temporairement des systèmes de fichiers (voir plus loin) ;
- **opt** contient les applications tierces ;
- **proc** est un répertoire virtuel (non physiquement existant sur disque) créé par le noyau. Il y place des fichiers d'information sur les processus, ainsi que des fichiers permettant de modifier certaines options du système (un peu comme la base de registres de Windows) ;
- **root** est le répertoire personnel de root ;
- **sbin** contient des commandes élémentaires d'administration ;
- **srv** contient les données des services (notamment réseau) hébergés par le système ;
- **sys** est un autre répertoire virtuel créé par le noyau, contenant des fichiers d'information/modification du système et des périphériques ;
- **tmp** contient des fichiers temporaires, pouvant être supprimés lors d'un redémarrage ;
- **usr** est la hiérarchie secondaire, contenant elle-même des répertoires `bin`, `lib`, `sbin`, ... qui sont peuplés d'éléments non essentiels du système, mais de la grande majorité des outils ;
- **var** contient des données variables, telles que les journaux du système, les files d'impression, les boîtes de réception des courriers électroniques, etc.

2. Il y a des exceptions pour certains systèmes de fichiers comme la majorité des clés USB ou cartes mémoires (voir plus loin).

3. Une bibliothèque est un fichier contenant du code et des informations auxquels les programmes peuvent faire appel.




## 2.C Références des fichiers

Il est capital de pouvoir désigner sans ambiguïté n'importe quel fichier de l'arborescence. Par exemple, dans l'arborescence d'allegro, comment désigner le fichier `main.cxx` ? Aussi, comment distinguer le fichier `toto` du répertoire `essais` et le répertoire `toto` de `home` ? De même pour les deux fichiers appelés `fic1` ?

Un élément de réponse est que **le nom d'un fichier n'a de sens que dans le répertoire dans lequel il apparaît**. Ainsi, sur la figure, `main.cxx` n'a de sens que dans le répertoire `src`, alors que `fic1` a un sens dans les deux répertoires `rep1` et `rep2` mais pas ailleurs. Avant de spécifier le nom du fichier visé, il faut donc indiquer, d'une manière ou d'une autre, le répertoire dans lequel celui-ci se trouve. Pour cela, le système admet deux possibilités :

- soit indiquer la **référence absolue** (ou chemin absolu) du fichier ;
- soit indiquer sa **référence relative** (ou chemin relatif).

 Ces deux références se distinguent par la présence d'un slash / **toujours** au début de la référence absolue mais **jamais** de la référence relative.

### 2.C.1 Référence absolue et chemin absolu

La référence absolue **commence forcément par un /**, appelé *slash* ou **barre oblique**, qui signifie que l'on va référencer le fichier à partir de la racine du système de fichiers. On indique alors les répertoires traversés sur la branche menant au fichier, en les séparant par un slash /. En ajoutant un / et le nom du fichier, on obtient enfin la référence absolue du fichier.

#### Exemple

Sur l'arborescence d'allegro (figure 2.1), la référence absolue du répertoire `essais` est :

`/home/cpb/public/unix/essais`

Puisqu'il s'agit d'un répertoire, on peut aussi ajouter un / à la fin, ce qui revient au même :

`/home/cpb/public/unix/essais/`

La référence absolue de son fichier `toto` est :

`/home/cpb/public/unix/essais/toto`

et puisqu'il ne s'agit pas d'un répertoire, on ne doit pas ajouter de / à la fin.

□

**i** Dans les documentations en anglais, le terme *path* (chemin) est parfois employé pour dire référence, et parfois il sert à désigner un répertoire qui mène à autre chose. Il faut l'interpréter selon le contexte. En français, cela conduit à distinguer deux emplois de chemin :

- le chemin (absolu ou relatif) **du** fichier `toto` est sa référence (absolue ou relative) ;
- le chemin (absolu ou relatif) **vers** (ou menant à) `toto` est le chemin (absolu ou relatif) **du** répertoire `essais`.

#### Exercice

1. Quelle est la référence absolue du répertoire `toto` de `home` ?
2. Quelle est la référence absolue du fichier `fic1` de `rep1` ?

## 2.C.2 Répertoire de travail, référence relative et chemin relatif

Sur tous les systèmes d'exploitation, un programme s'exécutant (processus) possède toujours un **répertoire de travail** (*working directory*). C'est le cas d'un shell `bash` interactif (où l'on tape des commandes). Lorsqu'on ouvre un terminal, le répertoire de travail du shell qui s'y exécute est initialement le répertoire d'accueil de l'utilisateur (voir section 1.D page 3).

Le répertoire de travail sert de base aux références relatives des fichiers, comme la racine sert de base aux références absolues. Dit autrement, une référence relative (ou chemin relatif) indique la référence du fichier à partir du répertoire de travail. En particulier, si le fichier ciblé se trouve dans le répertoire de travail, sa référence relative est simplement son nom.



**Au contraire de la référence absolue, la référence relative ne commence pas par un slash `/`.**

### Exemple

Toujours dans l'arborescence d'allegro (figure 2.1), supposons que le répertoire de travail soit `/home/toto` (le répertoire d'accueil de l'utilisateur `toto`). La référence relative de son fichier `.profile` est simplement `.profile`. Celle du fichier `main.cxx` est `projets/bidule/src/main.cxx`. On note qu'aucune ne commence par `/`, sinon ce serait une référence absolue (et incorrecte).



### Exercice

Supposons que le répertoire de travail soit `/home/cpb` :

1. quelle est la référence relative du répertoire `tp` ?
2. et celle du fichier `fic1` de `rep2` ?



Pour désigner un fichier dans une commande on utilise à sa guise une référence absolue ou relative.



Les systèmes permettent aux processus de changer de répertoire de travail (se déplacer dans l'arborescence). La commande `cd` (voir section 2.E.1 page 20) permet de changer celui du shell.

## 2.C.3 Le répertoire `.` (*point*)

Dans tout répertoire, il existe un répertoire particulier dont le nom est `.` (*point*). Ce nom désigne le répertoire qui le contient. Ainsi, ce n'est pas vraiment un répertoire mais disons un raccourci<sup>4</sup> vers son répertoire parent.

Pour illustrer ce concept, la figure 2.2 présente un extrait de l'arborescence d'allegro en faisant apparaître différents répertoires `.` et les répertoires parents qu'ils désignent.

4. Plus exactement, c'est un autre nom (un lien dur) pour son répertoire parent, et non pas un vrai raccourci (lien symbolique). Nous verrons cette différence dans la section 14.A.

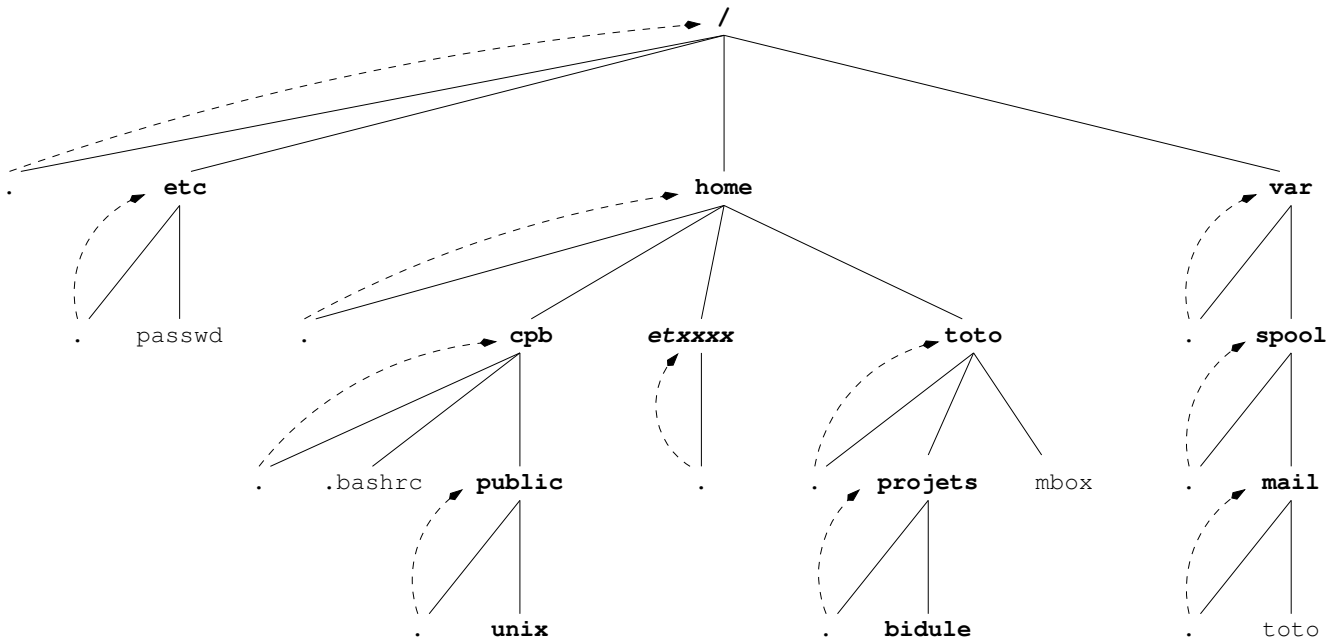



FIGURE 2.2 – Les répertoires `.` désignent leur répertoire parent. Tous les répertoires possèdent le leur.

 Ainsi, on peut désigner le répertoire de travail dans un shell par un simple point.

### Exemple

La commande **cp** (vue en section 2.E.1) copie des fichiers sources dans une destination. Supposons que le répertoire de travail soit `/home/toto`. Pour copier le fichier `/etc/passwd` dans le répertoire de travail, il faut exécuter :

```
$ cp /etc/passwd .
```

➡ le `$` représente le prompt, **cp** la commande, `/etc/passwd` est la référence absolue du fichier source, et la destination est le répertoire de travail (`.`), donc `/home/toto`.

□

❗ Remarquons que la référence (absolue et relative) d'un fichier n'est pas unique. Avec `/home/toto` comme répertoire de travail, les références :

```
mbox
./mbox
../mbox
/home/./toto/./mbox
../home/toto/mbox
```

désignent toutes le fichier `mbox` de `toto`.

❗ Nous verrons que le répertoire *point* s'avère aussi utile pour exécuter les programmes du répertoire de travail.

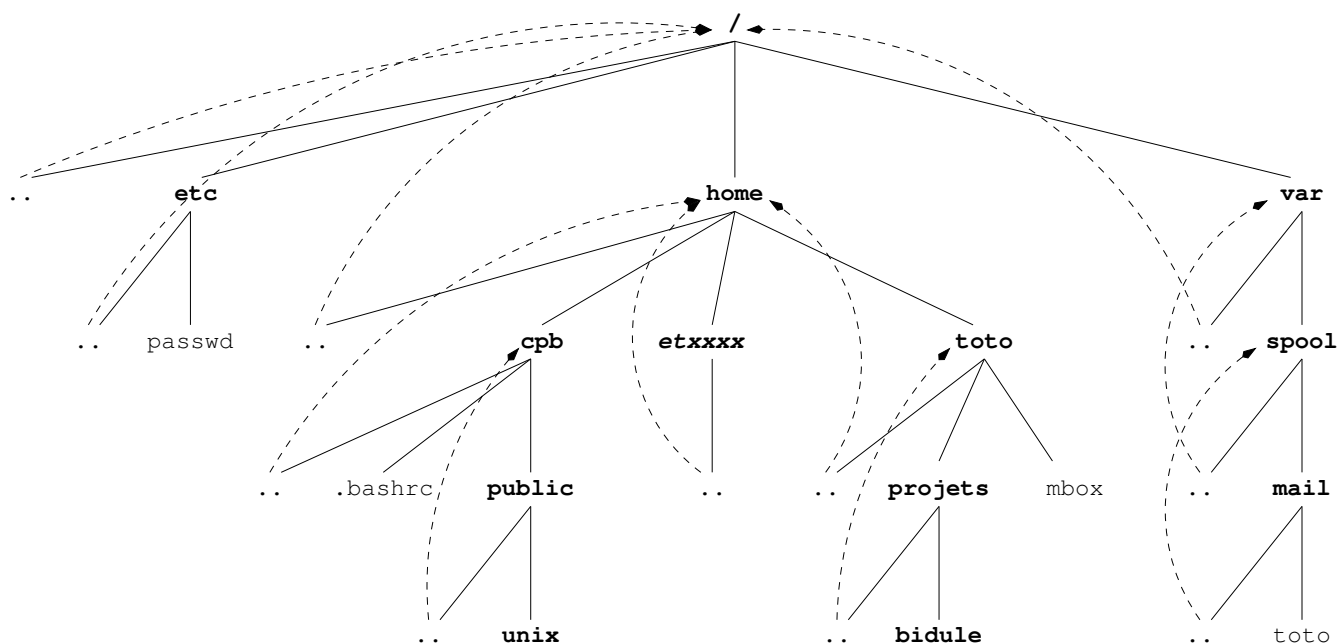


FIGURE 2.3 – Les répertoires `..` désignent leur répertoire grand-parent. Tous les répertoires possèdent le leur.

### 2.C.4 Le répertoire `..` (*point-point*)

À l'instar du répertoire `.`, il existe dans tout répertoire un répertoire `..` (*point-point*) qui désigne le répertoire parent de son répertoire parent (en somme son répertoire grand-parent).

La figure 2.3 présente un extrait de l'arborescence d'allegro en faisant apparaître différents répertoires `..` et les répertoires qu'ils désignent.

 On remarque que le répertoire `..` de la racine désigne la racine.

Dans le répertoire `/home/toto`, le répertoire `..` est donc en réalité le répertoire `/home`.

### Exercice

En supposant que le répertoire de travail soit `/home/toto`, quelle est la référence relative de :

1. `etxxxx` de `home` ?
2. `toto` de `mail` ?

## 2.D Disques, partitions et systèmes de fichiers

Sous Unix, les unités de stockage (ou mémoires secondaires) — (partitions des) disques durs (internes ou externes), clés USB, cartes mémoire, CD-ROM, etc. — sont associées à une partie de l'arborescence et apparaissent comme des répertoires. Plus exactement, **une partition est montée sur un répertoire existant** (quelconque) appelé **point de montage**. L'arborescence contenue dans la partition est alors accessible via ce répertoire. La commande externe **mount** permet d'afficher les montages utilisés et de monter des partitions. La commande externe **umount** permet de démonter des partitions.

### Exemple

Par exemple, *allegro* (cf. figure 2.1) utilise 5 partitions : une pour `/home`, une pour `/tmp`, une pour `/usr`, une pour `/var` et une autre pour la racine (en excluant `/home`, `/tmp`, etc.). On le vérifie avec **mount** :

```
$ mount
/dev/sda1 on / type ext3 (rw,errors=remount-ro)
/dev/sda9 on /home type ext3 (rw,usrquota)
/dev/sda8 on /tmp type ext3 (rw)
/dev/sda5 on /usr type ext3 (rw)
/dev/sda6 on /var type ext3 (rw,usrquota)
tmpfs on /lib/init/rw type tmpfs (rw,nosuid,mode=0755)
proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
udev on /dev type tmpfs (rw,mode=0755)
tmpfs on /dev/shm type tmpfs (rw,nosuid,nodev)
devpts on /dev/pts type devpts (rw,noexec,nosuid,gid=5,mode=620)
fusectl on /sys/fs/fuse/connections type fusectl (rw)
```

➡ Les 5 premières lignes correspondent aux partitions dont les points de montage sont mis en évidence en gras. Les autres lignes listent les systèmes de fichiers fictifs (pas des partitions de disques) créés par le noyau, parmi lesquels on reconnaît `/proc`, `/sys` et `/dev`.

□

Au minimum, un système Unix requiert deux partitions dont il se sert dès le démarrage : une pour la racine, et une autre —appelée **partition d'échange** ou *swap*— qui est une partition spéciale ne figurant pas dans l'arborescence et qui est réservée au noyau pour la gestion mémoire. Comme pour *allegro*, les administrateurs préfèrent souvent utiliser des partitions supplémentaires. Notamment, il est vivement conseillé de créer une partition pour `/home`. Les données des utilisateurs se trouvent ainsi séparées du système, ce qui est pratique en cas de réinstallation de ce dernier.

❗ Cela vaut aussi pour les systèmes Windows, où il vaut mieux utiliser une partition à part pour les documents des utilisateurs qui ne seront ainsi pas touchés par une éventuelle réinstallation du système.

En général, les partitions des disques durs internes (ou équivalents) sont montées pendant le démarrage du système. Cela se configure en éditant le fichier `/etc/fstab` qui indique au système les partitions à monter et leur point de montage. Sur un Linux récent, les supports amovibles (CD/DVD, disquettes, clés USB, cartes mémoires, etc.) sont montés automatiquement dans un sous-répertoire de `/media` lorsqu'ils sont insérés.

❗ De même, il n'existe pas de "lecteur réseau" : les espaces de stockage distants doivent aussi être montés sur des répertoires. Par exemple, un partage de fichiers Windows doit être monté sur un répertoire (via les utilitaires SAMBA), et en apparence rien ne le distingue des autres répertoires de l'arborescence.

Le noyau fait apparaître les unités de stockage et leurs partitions comme des fichiers de périphériques en mode bloc dans `/dev` (le répertoire des périphériques). Leur nom dépend du type d'unité de stockage, de leur nombre, et du numéro de partition. Le noyau fournit en outre le fichier `/proc/partitions` qui en donne la liste. La table des partitions peut être affichée (et modifiée) avec l'utilitaire **fdisk**.

### Exemple

*allegro* ne possède qu'un seul disque dur, dont le nom est `/dev/sda` (un deuxième disque dur de même type serait nommé `/dev/sdb`, un troisième `/dev/sdc`, etc.). Il a plusieurs partitions, ainsi que l'indique la commande **fdisk** (en root) :

```
# fdisk -l
```



```

Disk /dev/sda: 42.9 GB, 42949672960 bytes
255 heads, 63 sectors/track, 5221 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk identifier: 0x000102a9

```

Device	Boot	Start	End	Blocks	Id	System
/dev/sda1	*	1	43	340992	83	Linux
Partition 1 does not end on cylinder boundary.						
/dev/sda2		43	5222	41598977	5	Extended
/dev/sda5		43	1137	8787968	83	Linux
/dev/sda6		1137	1502	2928640	83	Linux
/dev/sda7		1502	1966	3729408	82	Linux swap / Solaris
/dev/sda8		1967	2015	389120	83	Linux
/dev/sda9		2015	5222	25759744	83	Linux

⇒ L'affichage commence par des informations sur le disque (/dev/sda), suivies de sa table des partitions. La partition /dev/sda1 est une partition primaire ; la partition /dev/sda2 est une partition étendue nécessaire pour créer les 5 partitions (dites secondaires) /dev/sda5 à /dev/sda9. La partition /dev/sda7 est la partition swap.

Ces partitions apparaissent dans le contenu du fichier /proc/partitions :

```

major minor #blocks name
8         0    41943040 sda
8         1      340992 sda1
8         2           1 sda2
8         5    8787968 sda5
8         6    2928640 sda6
8         7    3729408 sda7
8         8     389120 sda8
8         9   25759744 sda9

```

On peut vérifier que le disque /dev/sda et ses partitions correspondent à des fichiers de périphériques en mode bloc, à l'aide de la commande **file** :

```

$ file /dev/sda /dev/sda1 /dev/sda2 /dev/sda5 /dev/sda6 /dev/sda7 /dev/sda8 /dev/sda9
/dev/sda: block special
/dev/sda1: block special
/dev/sda2: block special
/dev/sda5: block special
/dev/sda6: block special
/dev/sda7: block special
/dev/sda8: block special
/dev/sda9: block special

```

□

Pour être utilisable et montée, une partition doit avoir été **formatée** pour accueillir un **système de fichiers**. Un système de fichiers est ce qui organise une partition pour y placer des fichiers. De nombreux systèmes de fichiers existent. Chacun a ses limitations, dont le nombre maximal de fichiers, la taille maximale des fichiers et leur nommage. Les anciennes versions de Windows (et de MS-DOS) utilisaient le système **FAT**, très limité dans le nommage des fichiers (8 caractères plus 3 pour l'extension sans distinction de casse), leur taille et la taille des partitions. La version **FAT32** est celle majoritairement utilisée pour formater les clés USB et les cartes mémoire. Elle admet des noms de fichiers d'au plus 255 caractères (toujours sans distinction de casse) faisant au plus 4 Gio et gère des partitions d'au plus 2 Tio (8 Tio en théorie).

❗ Historiquement en informatique, on a utilisé à tort les préfixes «kilo», «mega», «giga», «tera», etc. pour exprimer des quantités d'octets, reposant sur des puissances de 2. Par exemple, 1 Ko représentait 1024 octets ( $2^{10}$ ). Pour être conforme aux normes internationales des unités, ces préfixes ne devraient être utilisés que pour les puissances de 10. Ainsi, 1 Ko devrait désormais correspondre à 1000 octets ( $10^3$ ). Pour les unités historiques en puissances de 2, il faudrait maintenant utiliser les préfixes «kibi», «mébi», «gibi», «tébi», etc., où 1 Kio = 1024 octets ; 1 Mio (mébioctet) = 1024 Kio ; 1 Gio (gibioctet) = 1024 Mio ; 1 Tio (tébioctet) = 1024 Gio. Malheureusement, cet usage n'est pas encore très répandu dans le monde informatique, ce qui prête à confusions.

Les versions récentes de Windows utilisent prioritairement **NTFS**, autorisant (en pratique) des fichiers d'au plus 16 Tio sur des partitions d'au plus 256 Tio. Sous Linux, le système de fichiers le plus utilisé actuellement est **ext3**, autorisant des fichiers d'au plus 2 Tio sur des partitions d'au plus 32 Tio. Mais le noyau prend en charge un grand nombre de systèmes de fichiers (comprenant NTFS et FAT32), certains bien moins limités que **ext3**.

L'une des caractéristiques des systèmes de fichiers, dont dépendent la taille maximale des fichiers et de la partition, est la taille des blocs (logiques) de données utilisée. Un bloc logique est la plus petite partie du disque que le système de fichiers alloue pour y stocker le contenu (ou une partie) d'un fichier. Par exemple, si le bloc logique est de 1 Kio, alors un fichier de 3 octets occupe 1 Kio sur disque ! Un fichier de 1200 octets occupe 2 blocs logiques. Souvent, les systèmes de fichiers admettent plusieurs tailles de bloc logique. Lors du formatage d'une partition, l'administrateur choisit la taille de bloc logique qui correspond à ses besoins. Pour **ext3**, il peut choisir entre 1, 2, 4 ou 8 Kio. Néanmoins, la taille du bloc logique est dépendante du matériel, car un disque est lui-même composé de blocs physiques et la taille d'un bloc logique doit être un multiple de celle d'un bloc physique. Historiquement de 512 octets, les blocs physiques ont été augmentés par les fabricants avec la taille des disques, et font souvent maintenant 4 Kio.

## 2.E Manipulations élémentaires des fichiers

### 2.E.1 cd : changer de répertoire de travail

**cd** (*change directory*) est la commande interne permettant de changer de répertoire de travail. Dans le jargon informaticien, on dit aussi « *se placer dans un répertoire* » ou « *aller dans un répertoire* ».

#### Synopsis

**cd** [répertoire]

En exécutant **cd** sans argument, le répertoire de travail est positionné sur le répertoire d'accueil de l'utilisateur. Si *répertoire* est le caractère `-` (*tiret*), alors on revient au répertoire de travail précédent (celui avant la précédente utilisation de **cd**). Sinon, *répertoire* doit être une référence à un répertoire existant et accessible, qui devient le répertoire de travail.

#### Exemples

Supposons que l'utilisateur *cpb* travaille sur *allegro*. Son répertoire de travail est `/home/cpb` :

```
cpb@allegro:~$ cd public
```

➡ *cpb se déplace dans son répertoire public. On remarque que le prompt change*

```
cpb@allegro:~/public$ cd unix
```

➡ *va dans le répertoire unix de public*

```
cpb@allegro:~/public/unix$ cd /usr/bin
```

⇨ *utilise un chemin absolu pour aller dans /usr/bin*

```
cpb@allegro:/usr/bin$ cd -  
/home/cpb/public/unix
```

⇨ *revient au répertoire précédent*

❗ Lorsque l'argument n'est pas un chemin relatif ou absolu (par exemple - (tiret) ou en utilisant les facilités de **CDPATH**), le répertoire de destination est affiché par **cd** avant de s'y déplacer.

```
cpb@allegro:~/public/unix$ cd -  
/usr/bin  
cpb@allegro:/usr/bin$
```

⇨ *va à nouveau au répertoire précédent qui est /usr/bin*

```
cpb@allegro:~/public/unix$ cd  
cpb@allegro:~$
```

⇨ *se place par défaut dans son répertoire d'accueil (/home/cpb)*

□

## Exercice

Supposons que l'utilisateur toto travaille sur allegro. Son répertoire de travail (et d'accueil) est /home/toto. Quelles sont les commandes (on suppose qu'elles sont exécutées dans l'ordre) :

1. Permettant d'aller dans le répertoire `projets/bidule` (`projets` est contenu dans le répertoire personnel de l'étudiant toto) ?
2. Permettant d'aller dans le répertoire `home` (utiliser une référence absolue) ?
3. Permettant d'aller dans le répertoire `public` de `cpb` ?
4. Revenir au répertoire `home` ?
5. Aller dans le répertoire `etc` (utiliser une référence relative) ?
6. Revenir à son répertoire d'accueil ?

❗ (À sauter en première lecture) Il est possible de définir la variable d'environnement **CDPATH** qui facilite la navigation dans les répertoires. Cette variable contient une liste de répertoires (de la même manière que **PATH**). Dans ce cas, si *répertoire* est une référence relative ne correspondant pas à un répertoire existant, il sera recherché à partir des répertoires indiqués dans cette liste (dans leur ordre d'apparition).

## 2.E.2 pwd : afficher le répertoire de travail

La commande interne **pwd** (*print working directory*) affiche la référence absolue du répertoire de travail. Il n'y a pas d'argument à cette commande.

### Exemples

Supposons que /home/toto soit le répertoire de travail et d'accueil :

```
toto@allegro:~$ cd projets/bidule
toto@allegro:~/projets/bidule$ pwd
/home/toto/projets/bidule
```

⇨ le prompt affiche généralement une information sur le répertoire de travail mais cela dépend de la configuration de *bash*. De plus, **pwd** affiche le chemin complet (absolu), qui est plus précis.

```
toto@allegro:~/projets/bidule$ cd
toto@allegro:~$ pwd
/home/toto
toto@allegro:~$ cd /home
toto@allegro:/home$ pwd
/home
```

□

❗ (À sauter en première lecture) **pwd** admet l'option **-P** affichant la référence absolue ne comportant que les répertoires physiques (fichiers répertoires) et pas les liens symboliques (qui sont déréférencés).

### 2.E.3 ls : afficher le contenu des répertoires

La commande externe **ls** (*list*) affiche le contenu (ou une partie) d'un (ou plusieurs) répertoire(s). Elle admet de (très) nombreuses options dont certaines sont expliquées ici. Dans ce qui suit, le terme *fichier* est à prendre au sens large.

#### Synopsis

```
ls [-adFhilrRStuU] [--color=(none|always|auto)] {référence}
```

Sans option ni argument, **ls** affiche le contenu (non caché) du répertoire de travail, trié par ordre alphabétique.

✍ Les fichiers cachés sont les fichiers dont le nom commence par un point. Ainsi, `.bashrc` et `.bash_profile` sont deux fichiers cachés.

❗ Les répertoires `.` (*point*) et `..` (*point-point*) sont des fichiers (répertoires) cachés.

L'option **-a** (*all*) permet d'afficher aussi les fichiers cachés.

L'option **-l** (*long*) demande d'afficher des informations détaillées sur les fichiers écrits comme leur type, les permissions associées à ces fichiers, le nom du propriétaire, son groupe, sa taille, la date de sa création et enfin son nom (voir exemples).

Si une *référence* est un répertoire, **ls** affiche par défaut son contenu. L'option **-d** demande à afficher le répertoire lui-même. La combinaison **-ld** permet d'obtenir des informations détaillées sur les répertoires en arguments. Si une *référence* n'est pas un répertoire, la commande s'applique directement à elle (par exemple, affiche ses informations détaillées). Ainsi, l'option **-d** n'a aucun effet sur les arguments autres que les répertoires. Si une *référence* ne correspond à aucun fichier, **ls** affiche un message d'erreur pour cette *référence*.

L'option **-R** (*Récuratif*) demande un affichage récursif des répertoires en arguments. Leur arborescence est alors affichée.

L'option **-h** (humain) s'utilise en complément de **-l** et demande que les tailles soient affichées dans un format humainement lisible (8.2K, 465K, 1,4M, ...).

L'option **-i** (*inode*) demande d'afficher aussi le numéro d'inode (voir section 14.A page 191) du fichier à gauche de ses informations.

L'option **-F** demande de rajouter un caractère en fin de nom de fichier selon son type (**\*** pour un fichier régulier exécutable, **/** pour un répertoire, **@** pour un lien symbolique, **|** pour un tube nommé).

L'option **--color=** permet de demander ou non un affichage avec couleurs : **none** désactive l'affichage avec couleurs, **always** demande que les codes de couleurs soient toujours écrits, et **auto** (mais aussi **tty**) demande un affichage avec couleurs uniquement si la sortie est un terminal (c.-à-d. si l'affichage se fait à l'écran). Si la couleur est exploitée, alors les noms des fichiers sont affichés avec une couleur correspondant à leur type ou leur extension (bleu foncé pour les répertoires, vert pour les fichiers réguliers exécutables, etc.). Ces associations de couleurs sont indiquées par la variable d'environnement **LS\_COLORS** (on verra les variables d'environnement plus tard).

## Ordre d'affichage des fichiers

Les fichiers sont normalements affichés en étant triés par leur nom (ordre lexicographique croissant). Plusieurs options provoquent un ordre d'affichage différent :

- L'option **-S** demande à ce que le tri se fasse par la taille, par ordre décroissant ;
- L'option **-t** demande à ce que le tri se fasse par date de modification (du plus récemment modifié au plus ancien) ;
- L'option **-u** demande à ce que le tri se fasse sur la date d'accès (du plus récemment accédé/ouvert au plus ancien). Combinée à **-lt**, trie selon la date d'accès et affiche cette date à la place de la date de modification. Mais combinée à **-l** seulement, affiche la date d'accès et trie selon l'ordre lexicographique ;
- L'option **-U** désactive tous les tris et affiche les fichiers selon leur ordre d'apparition dans le répertoire ;
- Dans tous les cas (hors option **-U**), l'option **-r** (*reverse*) inverse l'ordre de tri.

❗ Historiquement, l'ordre lexicographique suivait celui des codes ASCII. Désormais, l'ordre dépend de la **locale** de l'utilisateur (sa langue et son jeu de caractère) et suit celui en usage pour sa locale. Cela conduit à donner le même ordre aux majuscules, minuscules et caractères accentués, et à ignorer les blancs, tirets, etc. Pour forcer **ls** à utiliser l'ordre ASCII, il faut définir la variable d'environnement **LC\_ALL** ainsi :

```
$ export LC_ALL=C
```

## Exemples

```
~/repl$ pwd
/home/cpb/public/unix/essais/repl
~/repl$ ls
fic1 lien truc un-rep
~/repl$ ls -a
. .. fic1 .fic_cache lien truc un-rep
```

⇒ les 3 fichiers cachés `..`, `...` et `.fic_cache` apparaissent

```
~/repl$ ls -aF
./  ../  fic1  .fic_cache  lien@  truc*  un-rep/
```

⇒ les caractères `/`, `@` et `*` en fin de nom de fichiers donnent des précisions sur leur type

```
~/repl$ ls -l
total 80
-rw-r--r--  1 cpb      prof      0 août 31 18:35 fic1
lrwxrwxrwx  1 cpb      prof      4 août 31 19:39 lien -> fic1
-rwxr-xr-x  1 cpb      prof    69708 août 31 19:42 truc
drwxr-xr-x  3 cpb      prof    4096 août 31 19:41 un-rep
```

⇒ affichage des informations détaillées sur les fichiers contenus dans le répertoire de travail

### Comprendre l'option -d

L'expérience montre que cette option est souvent mal comprise, alors qu'elle est pourtant simple. Les deux exemples qui suivent devraient permettre de la clarifier.

```
~/repl$ ls -l un-rep
total 12
-rw-r--r--  1 cpb      prof    123 août 31 18:35 unfic
-rw-r--r--  1 cpb      prof    432 août 31 19:39 unautrefic.txt
drwxr-xr-x  2 cpb      prof    4096 août 31 19:41 autrerep
```

⇒ l'argument de `ls` étant un répertoire (`un-rep`), c'est son contenu qui est affiché (avec détails à cause de l'option `-l`)

```
~/repl$ ls -ld un-rep
drwxr-xr-x  3 cpb      prof    4096 août 31 19:41 un-rep
```

⇒ l'ajout de l'option `-d` demande de ne pas afficher le contenu de `un-rep` mais les informations sur lui-même. C'est donc les détails sur `un-rep` qui sont affichés. Simple, non ?

### Analyse des informations détaillées de ls

```
~/repl$ ls -al
total 92
drwxr-xr-x  3 cpb      prof    4096 août 31 19:43 .
drwxr-xr-x  4 cpb      prof    4096 août 31 19:43 ..
-rw-r--r--  1 cpb      prof      0 août 31 18:35 fic1
-rw-r--r--  1 cpb      prof      4 août 31 19:40 .fic_cache
lrwxrwxrwx  1 cpb      prof      4 août 31 19:39 lien -> fic1
-rwxr-xr-x  1 cpb      prof   69708 août 31 19:42 truc
drwxr-xr-x  3 cpb      prof    4096 août 31 19:41 un-rep
```

Attardons-nous sur les lignes rendues par la commande `ls -al`. Elles donnent beaucoup de renseignements concernant les fichiers contenus dans `repl` (le répertoire de travail). Ces lignes sont toujours écrites de la même manière. En effet, `ls` avec l'option `-l` écrit, pour chaque répertoire en arguments (ou à défaut pour le répertoire de travail), une première ligne « total ... » qui indique le nombre de blocs occupés par les fichiers affichés. Dans ce contexte, la taille d'un bloc est indépendante du système de fichiers et est paramétrable. Elle est généralement 512 Kio ou 1024 Kio. Les lignes qui suivent écrivent dans l'ordre les éléments suivants :

- un caractère indiquant le type du fichier :

- ◇ **-** pour un fichier ordinaire (ou régulier) ;
- ◇ **d** pour un répertoire (directory) ;
- ◇ **l** pour un fichier spécial de type lien (link) symbolique (voir section 14.A) ;
- ◇ **c** pour un fichier (spécial) de périphérique en mode caractère, tel qu'un terminal ;
- ◇ **b** pour un fichier (spécial) de périphérique en mode bloc, tel qu'une partition d'un disque dur ;
- ◇ **s** pour un fichier spécial de type socket (communication entre processus) ;
- ◇ **p** pour un tube (pipe) nommé (communication entre processus).

Ainsi *un-rep* est un répertoire, *fic1* et *.fic\_cache* sont des fichiers ordinaires et *lien* est un lien symbolique ;

- 9 caractères indiquant les permissions du fichier. Pour *fic1*, elles sont **rw-r-r-** et pour *un-rep*, ce sont **rwxr-xr-x** (ceci sera détaillé dans la section 2.F page 33) ;
- le nombre de liens physiques (voir section 14.A) sur le fichier : 1 pour *fic1* et 2 pour *un-rep*.
- le nom de l'utilisateur propriétaire du fichier (voir section 2.F). Ici, les fichiers appartiennent tous à *cpb*.
- le nom du groupe auquel appartient le fichier. Ici, ils sont tous du groupe *prof*.
- la taille en octets du fichiers ;
- la date et l'heure de sa dernière modification ;
- son nom.

```
$ ls -l /dev/sda9 /dev/tty1 /tmp/.X11-unix/X0
brw-rw---- 1 root disk 8, 9 29 août 09:38 /dev/sda9
crw----- 1 root root 4, 1 29 août 09:39 /dev/tty1
srwxrwxrwx 1 root root 0 29 août 09:39 /tmp/.X11-unix/X0
```

⇒ quelques exemples de fichiers spéciaux (partition, terminal et socket).

```
~/repl$ ls -alh
total 92
drwxr-xr-x 3 cpb prof 4K août 31 19:43 .
drwxr-xr-x 4 cpb prof 4K août 31 19:43 ..
-rw-r--r-- 1 cpb prof 0 août 31 18:35 fic1
-rw-r--r-- 1 cpb prof 4 août 31 19:40 .fic_cache
lrwxrwxrwx 1 cpb prof 4 août 31 19:39 lien -> fic1
-rwxr-xr-x 1 cpb prof 68.1K août 31 19:42 truc
drwxr-xr-x 3 cpb prof 4K août 31 19:41 un-rep
```

⇒ Avec l'option **-h**, les tailles des fichiers sont plus faciles à lire.

□

❗ Bien souvent, dans les shells la commande **ls** est en fait un **raccourci** (voir les *alias* à la section 9.A.1) pour la commande **ls --color=auto**. Aussi, il existe souvent un alias **ll** comme raccourci pour **ls -l**. Ces alias éventuels sont habituellement créés au démarrage du shell (voir section 9.D).

## 2.E.4 mkdir : créer des répertoires

**mkdir** (*make directory*) est une commande externe permettant de créer les répertoires indiqués en arguments.

### Synopsis



```
mkdir [-vp] référence {référence}
```

Chaque *référence* est la référence d'un répertoire à créer. Si *référence* contient des slashes, alors les répertoires du chemin doivent exister à moins de spécifier l'option **-p** (pour parents) auquel cas ils sont aussi créés. L'option **-v** (mode verbeux ou bavard) demande d'afficher un message pour chaque répertoire créé. Les répertoires sont créés dans l'ordre d'apparition sur la ligne de commande.

### Exemples

```
~/essais$ mkdir rep1 rep2
```

⇒ crée les deux répertoires *rep1* et *rep2*

```
~/essais$ mkdir -v rep1/rep1_1/rep1_1_1 rep1/rep1_1
```

```
mkdir: Ne peut créer le répertoire 'rep1/rep1_1/rep1_1_1'.: No such file or directory
mkdir: répertoire créé 'rep1/rep1_1'
```

⇒ impossible de créer *rep1/rep1\_1/rep1\_1\_1* car *rep1/rep1\_1* n'existait pas au moment de sa demande de création (mais existe à la fin de la commande)

```
~/essais$ mkdir -v rep1/rep1_1/rep1_1_1 rep1/rep1_1/rep1_1_1/rep1_1_1_1
```

```
mkdir: répertoire créé 'rep1/rep1_1/rep1_1_1'
mkdir: répertoire créé 'rep1/rep1_1/rep1_1_1/rep1_1_1_1'
```

⇒ cette fois les répertoires sont créés dans l'ordre

```
~/essais$ mkdir -v rep2/rep2_1/rep2_1_1
```

```
mkdir: Ne peut créer le répertoire 'rep2/rep2_1/rep2_1_1'.: No such file or directory
```

⇒ encore une fois le répertoire *rep2/rep2\_1* n'existant pas, on ne peut pas y créer *rep2\_1\_1*

```
~/essais$ mkdir -v -p rep2/rep2_1/rep2_1_1
```

```
mkdir: répertoire créé 'rep2/rep2_1'
mkdir: répertoire créé 'rep2/rep2_1/rep2_1_1'
```

⇒ l'option **-p** permet de créer à la fois *rep2/rep2\_1* et *rep2/rep2\_1/rep2\_1\_1*. Cela a le même effet que de taper `mkdir -v rep2/rep2_1 rep2/rep2_1/rep2_1_1`

```
~/essais$ cd rep2
```

```
~/essais/rep2$ mkdir ../rep1/rep1_2
```

⇒ n'importe quelle référence absolue ou relative est valable.

□

## 2.E.5 rmdir : supprimer des répertoires

La suppression de répertoires est réalisée avec la commande externe **rmdir**.

### Synopsis

```
rmdir [-vp] référence {référence}
```



Si elle est possible (autorisée), la suppression est irréversible !!!

L'option **-v** demande d'afficher un message pour chaque répertoire supprimé. L'option **-p** demande de supprimer la totalité du chemin si *référence* n'est pas un nom simple. Ceci est réalisé en supprimant d'abord le répertoire en queue du chemin (c.-à-d. le répertoire le plus profond dans *référence*), puis celui au niveau inférieur, etc.





Les répertoires effacés par `rmdir` doivent être vides.

### Exercice

Un répertoire vide contient quand même quelque chose. Quoi ?



La commande `rm` (voir section 2.E.8 page 30) permet de supprimer des répertoires non vides.

### Exemples

```
~/essais$ rmdir rep1/rep1_1/rep1_1_1/rep1_1_1_1
```

⇒ Suppression du répertoire `rep1_1_1_1`

```
~/essais$ rmdir -v rep1/rep1_2
```

```
rmdir: destruction du répertoire, << rep1/rep1_2 >>
```

```
~/essais$ rmdir -v rep1
```

```
rmdir: destruction du répertoire, << rep1 >>
```

```
rmdir: 'rep1': Directory not empty
```

⇒ `rep1` ne peut être supprimé car il n'est pas vide (contient `rep1_1` qui lui même contient `rep1_1_1`)

```
~/essais$ rmdir -vp rep1/rep1_1/rep1_1_1
```

```
rmdir: destruction du répertoire, << rep1/rep1_1/rep1_1_1 >>
```

```
rmdir: destruction du répertoire, << rep1/rep1_1 >>
```

```
rmdir: destruction du répertoire, << rep1 >>
```

⇒ L'option `-p` peut s'avérer assez pratique...

```
~/essais$ rmdir -v rep2/rep2_1/rep2_1_1/ rep2/rep2_1 rep2
```

```
rmdir: destruction du répertoire, << rep2/rep2_1/rep2_1_1/ >>
```

```
rmdir: destruction du répertoire, << rep2/rep2_1 >>
```

```
rmdir: destruction du répertoire, << rep2 >>
```

⇒ ... car sans utiliser l'option `-p` le même travail est réalisé mais avec une commande beaucoup plus longue !




## 2.E.6 cp : copier des fichiers

La commande externe `cp` (*copy*) permet de copier des fichiers.

### Synopsis

```
cp [-adfirpv] source {source} destination
```

Les arguments *source* et *destination* sont des références. `cp` copie le(s) fichier(s) *source* dans *destination*. Sans l'option `-r`, les *sources* ne peuvent pas être des répertoires. Si plusieurs *sources* sont spécifiées, *destination* doit être un répertoire existant qui recevra alors une copie de chaque *source* (même nom). Si une seule *source* est spécifiée, *destination* peut être un nouveau nom que portera la copie (fichier ordinaire, spécial ou répertoire) de *source*.

 Lorsqu'un fichier (au sens large) est créé en effectuant une copie, celui-ci a pour propriétaire et pour groupe ceux de l'utilisateur (effectif) qui réalise la copie. Ses permissions sont celles du fichier d'origine moins celles du masque de création de fichier (voir **umask**, section 2.F.5 page 39).

Si *destination* est un fichier qui existe, **cp** ne le supprimera pas mais tentera de remplacer son contenu (l'écraser), ce qui nécessite le droit d'écriture sur la *destination*. Dans ce cas, même si la copie réussit, *destination* garde son propriétaire, son groupe et ses permissions. Seuls son contenu et sa date de modification changent. L'option **-f** demande que si ce droit est absent (ce qui provoquerait une erreur), alors tenter de supprimer *destination* avant de procéder à la copie. Si la suppression réussit, tout se passe comme si la copie créait un nouveau fichier.

Lorsque l'option **-i** (*interrogation*) est demandée, si la copie d'un fichier source dans la destination appropriée conduit à l'écrasement d'un fichier, une confirmation est demandée<sup>5</sup>.

L'option **-r** (*récuratif*) autorise la copie de répertoires et de tout ce qu'ils contiennent (leur arborescence, c'est à dire toutes les branches qu'ils contiennent). Dans ce cas, *destination* doit être un répertoire ou le nom que portera la copie de *source*.

L'option **-v** (*verbeux*) demande d'afficher un message pour chaque fichier copié.

L'option **-d** demande de copier les liens symboliques eux mêmes et non les fichiers qu'ils référencent.


Les options **-p** et **-a** sont principalement destinées à *root* : **-p** demande que la copie garde les mêmes propriétaire/groupe, permissions et dates que la *source*. L'option **-a** va plus loin et demande de préserver tout ce qui est possible du fichier *source*, tout en activant les options **-d** et **-r**.

## Exemples

```
$ cp fic1 fic1.sve
```

 création d'une copie de *fic1*, appelée *fic1.sve*, dans le répertoire de travail


```
$ cp /home/boitard/pastiche.txt .
```

 création d'une copie du fichier *pastiche.txt* dans le répertoire de travail. La copie aura aussi le nom *pastiche.txt*.

```
$ cp -v fic1 fic2.txt un-rep
```


```
'fic1' -> 'un-rep/fic1'
```

```
'fic2.txt' -> 'un-rep/fic2.txt'
```

 création d'une copie des fichiers *fic1* et *fic2.txt* dans le répertoire *un-rep* (qui doit déjà exister). Les copies auront le même nom que les originaux.

```
$ cp -v un-rep un-rep.copie
```

```
cp: omission du répertoire 'un-rep'
```

 il faut utiliser l'option **-r** pour copier des répertoires.

```
$ cp -vr un-rep un-rep.copie
```

```
'un-rep' -> 'un-rep.copie'
```

```
'un-rep/fic1' -> 'un-rep.copie/fic1'
```

```
'un-rep/fic2.txt' -> 'un-rep.copie/fic2.txt'
```

5. Si les options contradictoires **-i** et **-f** sont utilisées, la version **cp** de coreutils 6.10 donne la priorité à **-i**. Pour d'autres versions, c'est l'option la plus à droite dans la ligne de commandes qui est retenue.

```
$ ls -l fic1
-rwxr-xr-x    1 cpb      prof      13 sep 10 02:15 fic1
$ umask 267
$ cp fic1 fic4
$ ls -l fic4
-r-x--x---    1 cpb      prof      13 sep 10 02:20 fic4
```

⇒ cet exemple montre qu'un fichier créé par copie possède les permissions du fichier d'origine moins celles du masque de création de fichiers

□

## 2.E.7 mv : déplacer ou renommer un fichier

La commande externe **mv** (*move*) permet de déplacer un (ou plusieurs) fichier(s) ou de le renommer, ou les deux en même temps.

### Synopsis

```
mv [-ivf] source {source} destination
```

*source* doit être un fichier (ou un répertoire) existant. Il sera soit déplacé dans le répertoire *destination* si celui-ci existe, soit renommé en *destination*. Si plusieurs *sources* sont spécifiées, *destination* doit être une référence à un répertoire existant.

L'option **-i** (*interrogation*) demande confirmation lorsque le déplacement ou le renommage d'un fichier a pour effet d'en écraser un autre.

L'option **-v** (*verbeux*) affiche un message pour chaque déplacement ou renommage.

L'option **-f** (*force*) désactive la demande de confirmation avant écrasement.

### Permissions nécessaires au déplacement/renommage d'un fichier

Pour déplacer/renommer un fichier **dans une même partition**, il faut avoir le droit d'accéder aux répertoires source et destination et le droit de les modifier (écriture). Aucun droit sur le fichier lui-même n'est nécessaire.

Lorsque le déplacement/renommage se fait d'une partition à l'autre, un droit supplémentaire est nécessaire : celui de lecture du fichier *source*, car il faut copier son contenu sur la nouvelle partition avant de le supprimer.

### Cas de l'écrasement de fichiers par déplacement

Le déplacement de fichiers peut conduire à la suppression de fichiers existants. Il ne s'agit pas vraiment d'écrasement, car pour que le déplacement réussisse, les permissions du fichier qui sera supprimé n'ont aucune importance. Seules comptent les permissions sur le répertoire de destination qui doivent être l'écriture et l'exécution.

À noter que le déplacement d'un répertoire peut conduire de la même manière à la suppression d'un répertoire existant, mais celui-ci doit être vide. D'autre part, le déplacement d'un fichier ne peut pas conduire à la suppression d'un répertoire, et inversement.

### Exemples

```
$ mv pastiche.txt mon-pastiche.txt
```

⇒ *renomme `pastiche.txt` en `mon-pastiche.txt` si `mon-pastiche.txt` n'est pas un répertoire existant.*

```
$ mv un-rep repertoire
```

⇒ *renomme `un-rep` en `repertoire` si `repertoire` n'est pas un répertoire existant.*

```
$ mv -v repertoire un-rep
```

```
'repertoire' -> 'un-rep'
```

⇒ *renomme `repertoire` en `un-rep`*

```
$ mv -v un-rep/fic2.txt fichier
```

```
'un-rep/fic2.txt' -> './fic2.txt'
```

⇒ *déplace `fic2.txt` depuis `un-rep`, en le renommant `fichier` dans le répertoire de travail*

```
$ mv fichier un-rep ..
```

⇒ *déplace `fichier` et le `un-rep` dans le répertoire père du répertoire de travail*

```
$ mv -iv ../un-rep/fic1 un-rep.copie
```

```
mv: écraser 'un-rep.copie/fic1'? n
```

```
$ mv -iv ../un-rep/fic1 un-rep.copie
```

```
mv: écraser 'un-rep.copie/fic1'? y
```

```
'../un-rep/fic1' -> 'un-rep.copie/fic1'
```

□

## 2.E.8 rm : supprimer des fichiers

La suppression de fichiers est réalisée par la commande externe **rm**.

### Synopsis

```
rm [-ifrv] référence {référence}
```



**La suppression des fichiers passés en arguments est souvent irréversible (dépend du systèmes de fichiers).**

C'est pourquoi, il vaut mieux utiliser l'option **-i** (*interrogation*) qui demande confirmation pour chaque fichier à supprimer.

L'option **-r** (*recursif*) permet de supprimer toute l'arborescence des répertoires passés en arguments (c'est à dire leur contenu et eux-mêmes). Sans elle, il n'est pas possible de supprimer un répertoire (un message d'erreur serait écrit).

L'option **-f** (*force*) annule les demandes de confirmation si placée après l'option **-i**. Même les demandes de confirmation en cas de tentative de suppression d'un fichier dont l'utilisateur est propriétaire mais sur lequel il n'a pas les droits d'écriture<sup>6</sup>, sont annulées.

Enfin, l'option **-v** (*verbeux*) affiche un message pour chaque suppression réalisée.

6. Voir section 2.F page 33.

## 2.E.9 df : afficher l'occupation disque

La commande externe **df** (*disk free*) affiche l'occupation des systèmes de fichiers utilisés.

### Synopsis

**df** [-H] {référence}

Sans option ni argument, **df** affiche l'occupation de tous les systèmes de fichiers (partitions, lecteurs réseau) montés, y compris certains systèmes fictifs. Si des *références* sont indiquées, l'affichage est limité aux systèmes de fichiers les contenant.

Pour chaque système de fichiers présenté, les informations sont affichées sur 6 colonnes : le nom du système de fichiers, sa taille en blocs de 1 Kio, le nombre de blocs utilisés, le nombre de blocs encore disponibles, le pourcentage d'occupation disque, et son point de montage.

L'option **-H** demande un affichage des quantités plus humainement lisible.

### Exemples

```
$ df
Sys. de fichiers      1K-blocs  Utilisé    Dispo.  Uti%  Monté sur
/dev/sda1             330215    179381    133785   58%   /
tmpfs                 1557572      0    1557572    0%  /lib/init/rw
udev                 1553128    108    1553020    1%  /dev
tmpfs                 1557572      0    1557572    0%  /dev/shm
/dev/sda9             25355368   367336   23700048    2%  /home
/dev/sda8              376807    10295    347056    3%  /tmp
/dev/sda5             8649992   2696560   5514036   33%  /usr
/dev/sda6             2882592   1986924    749236   73%  /var
```

```
$ df /home/et1001
Sys. de fichiers      1K-blocs  Utilisé    Dispo.  Uti%  Monté sur
/dev/sda9             25355368   367336   23700048    2%  /home
```

⇒ limite l'affichage au système de fichiers contenant */home/et1001*


```
$ df -H /home/et1001
Sys. de fichiers      Taille  Util.  Disp.  Uti%  Monté sur
/dev/sda9             26G    377M    25G    2%  /home
```

⇒ les quantités sont plus lisibles avec l'option **-H**

□

## 2.E.10 du : calculer la taille de fichiers ou répertoires

**du** (*disk usage*) est la commande externe permettant de calculer l'occupation disque de fichiers ou répertoires. Elle est très pratique car elle vous permet de vérifier que vous n'arrivez pas près du quota, et accessoirement de repérer des répertoires anormalement volumineux (du fait d'une mauvaise manipulation, par exemple).

 On entend par taille d'un répertoire, la place occupée par son fichier (répertoire) ainsi que par toute son arborescence.

### Synopsis

**du** [-askbhc] {référence}

Sans option, **du** parcourt les arborescences des répertoires (ou fichiers) indiqués par *référence*, et affiche la taille en blocs de tous les répertoires rencontrés. Sans argument, le répertoire traité est le répertoire de travail.

L'option **-a** demande d'afficher aussi la taille chaque fichier rencontré. L'option **-s** (*summarize*) demande de n'afficher ces informations que pour les arguments *référence*. L'option **-k** demande que la taille soit calculée en Kilo-octets (1 Ko = 1024 octets), alors que **-b** (byte) demande un calcul en octets (*byte* veut dire octet en français). L'option **-h** demande un affichage plus clair pour les humains (en utilisant K pour Ko, M pour Mo ou G pour Go). Enfin, l'option **-c** demande l'affichage du total des occupations calculées pour les arguments *référence*.

### Exemples

```
$ du prog tpunix
44      prog/essais
8       prog/tp
1       prog/perso
2416    prog/projet
2845    prog
2       tpunix/rep1
1       tpunix/rep2
1       tpunix/rep3
8       tpunix
$ du -cs prog tpunix
2845    prog
8       tpunix
2853    total
```

□

## 2.E.11 quota : connaître la limite autorisée de son occupation disque

L'espace disque dont on dispose n'est pas infini. Lorsque plusieurs utilisateurs partagent le même espace de stockage, il est nécessaire de fixer une limite de l'occupation disque autorisée pour chacun. Cela s'appelle le **quota**. Sans quota, l'espace disque risquerait d'être saturé par un utilisateur, et rendrait le système difficilement utilisable pour les autres.

En général, l'administrateur fixe un quota propre à chaque système de fichiers entier (arborescence correspondant à une partition). Il peut être de deux types :

- occupation disque : limite le nombre de blocs utilisés sur cette partition ;
- nombre de fichiers : limite le nombre de fichiers créés sur la partition.

 Les deux types de quotas peuvent s'appliquer en même temps.

L'administrateur peut affecter un quota différent selon les utilisateurs et/ou leurs groupes. La commande externe **quota** permet de consulter l'état actuel des quotas.

### Synopsis

```
quota
quota -u utilisateur {utilisateur}
quota -g groupe {groupe}
```

Sans option ni argument, affiche un état des quotas de l'utilisateur courant. L'option **-u** demande l'affichage des quotas pour les utilisateurs listés. L'option **-g** demande ceux pour les groupes listés.

**i** Seul root peut consulter les quotas d'un autre utilisateur ou d'un groupe étranger. Les utilisateurs ne peuvent consulter que leurs quotas et ceux de leurs groupes.

### Exemple

```
toto$ quota
```

Quotas disque pour user toto (uid 6745) :

Système fichiers	blocs	quota	limite	sursis	fichiers	quota	limite	sursis
/dev/sda9	252	30000	30500		48	0	0	
/dev/sda8	22	1000	1500		1	0	0	

□

Les deux dernières lignes indiquent les quotas s'appliquant à l'utilisateur (ici toto) sur les disques mentionnés en première colonne. La seconde colonne indique le nombre de blocs utilisés par l'utilisateur sur chacun des disques. La troisième colonne indique la limite (en nombre de blocs) à partir de laquelle l'utilisateur recevra des messages d'avertissement d'approche de sa limite autorisée. La quatrième colonne indique l'occupation limite qui lui est imposée. S'il la dépasse, il risque de ne plus pouvoir se connecter au système. La colonne qui suit (*sursis*) indique combien de temps il lui reste pour régulariser son occupation disque (s'il y a écrit **none**, l'utilisateur est devenu *persona non grata*).

Les quatre colonnes qui suivent jouent le même rôle que les précédentes mais en termes de nombres de fichiers.

Sur l'exemple, on voit que toto utilise 252 blocs sur le disque `/dev/sda9`, qu'il sera averti lorsqu'il aura atteint 30000 blocs et qu'il ne doit pas dépasser 30500 blocs (soit plus de 30 Mo). De même, il utilise 22 blocs sur le disque `/dev/sda8`, sera averti lorsqu'il aura atteint 1000 blocs et sa limite est 1500 blocs. En revanche, il n'a pas de quota concernant le nombre de ses fichiers.



**Au département informatique, un quota s'applique sur allegro pour les utilisateurs (pas pour les groupes). Sur les postes de travail, l'espace de stockage personnel correspond à un espace de stockage hébergé sur les serveurs de l'université.**

## 2.F Propriété et permissions des fichiers

Les opérations qui peuvent être effectuées sur un fichier sont : la lecture, l'écriture et l'exécution. Mais a-t-on pour autant le droit de lire, de modifier ou d'exécuter n'importe quel fichier ? Bien sûr que non. Il existe un mécanisme permettant de protéger les fichiers des utilisateurs contre l'indiscrétion et la malveillance d'autrui.

Il repose sur les permissions des fichiers, qui sont un des concepts de base de la gestion d'utilisateurs multiples sous Unix. Il va permettre d'interdire/autoriser la lecture, l'écriture ou l'exécution de certains fichiers par certains utilisateurs.

### 2.F.1 Principe


À chaque fois qu'un utilisateur veut effectuer une opération sur un fichier, le système va vérifier que cette opération lui est permise. Cette vérification repose sur les 5 informations suivantes :

- l'utilisateur qui tente d'effectuer l'opération ;
- ses groupes ;
- le propriétaire du fichier ;
- le groupe du fichier ;
- les permissions (ou droits d'accès) du fichier.

Ces trois dernières informations, relatives au fichier, peuvent être visualisées avec la commande **ls** (option **-l**) vue précédemment. En général, le propriétaire d'un fichier est l'utilisateur qui l'a créé et son groupe est le groupe principal du propriétaire<sup>7</sup>. Les permissions du fichier sont composées de 3 parties, chacune s'adressant à une catégorie d'utilisateur :

- les droits du propriétaire ;
- les droits des membres du groupe du fichier ;
- les droits des autres utilisateurs.

**Les droits d'un utilisateur sur un fichier sont uniquement ceux de sa catégorie la plus spécifique** (propriétaire, sinon membre du groupe, sinon autre). L'opération demandée par l'utilisateur ne sera autorisée que s'il possède les droits qu'elle nécessite.

 On notera que le propriétaire d'un fichier peut très bien posséder moins de droits sur ce fichier qu'un autre utilisateur. Mais s'il peut accéder à ce fichier, il sera toujours en mesure de se donner les droits qu'il veut.

## 2.F.2 Distinction entre répertoires et fichiers

Cette section précise le rôle des permissions, qui diffère sensiblement entre les fichiers (même spéciaux) et les répertoires.

### 2.F.2.a Les droits pour un fichier

**lecture** pour en lire le contenu (ou, comme on le verra plus tard, de l'utiliser comme redirection de l'entrée standard) ;

**écriture** pour en modifier le contenu (ou de l'utiliser comme redirection des sorties s'il existe déjà), **mais n'a pas d'incidence pour sa suppression**<sup>8</sup> ;

**exécution** pour l'exécuter s'il s'agit d'un fichier binaire contenant du code exécutable<sup>9</sup>. S'il s'agit d'un fichier texte (contenant ce qu'on appelle un *script*), il faut aussi le droit de lecture pour l'exécuter<sup>10</sup>.

### 2.F.2.b Les droits pour un répertoire

**lecture** consulter son contenu (afficher la liste des fichiers qu'il contient), notamment par la commande **ls** ;

**écriture** modifier le répertoire, ce qui comprend l'ajout d'un fichier (de tout type), sa suppression (même sans droit sur ce fichier !) ou son renommage (sans changer de propriétaire). Le droit d'écriture ne sert à rien si on n'a pas aussi le droit d'exécution ;

7. Ce n'est pas toujours vrai car les commandes **chown** et **chgrp** permettent de changer ces informations (encore faut-il avoir le droit de le faire...). Ces commandes seront décrites page 41.

8. Supprimer un fichier sur lequel on n'a pas le droit d'écriture peut provoquer l'affichage d'un message d'avertissement et/ou une demande de confirmation.


9. Le plus souvent, un fichier contenant un code exécutable est au format ELF (*Executable and Linking Format*).


10. Car le fichier doit être lu pour être interprété.



**exécution** autoriser l'accès au répertoire. Son absence permet donc d'interdire l'accès à une partie de l'arborescence du système de fichiers. En effet, sans le droit d'exécution sur un répertoire, un utilisateur ne peut pas le "traverser" et ne peut alors pas :

- ◇ faire de ce répertoire son répertoire de travail, c'est à dire aller dans ce répertoire (en utilisant **cd**) ;
- ◇ référencer un fichier qui se trouve dans l'arborescence de ce répertoire ;
- ◇ obtenir des détails sur les fichiers contenus dans ce répertoire (notamment avec l'option **-l** de **ls**).

 Ainsi, les droits sur un fichier ne sont effectifs pour un utilisateur que s'il possède le droit d'exécution sur le chemin qui y mène. Sinon, le fichier lui est inaccessible.

 L'utilisation assez courante de l'option **--color=auto** de **ls** requiert l'obtention des détails sur les fichiers pour les afficher avec des couleurs, et donc aussi du droit d'exécution. Cette option est généralement activée par défaut par l'intermédiaire d'*alias* (voir section 9.A page 101).

### 2.F.3 Connaître ses droits sur un fichier

Pour connaître ses droits sur un fichier, il faut utiliser une commande qui les indique sous une forme ou une autre. La plus couramment utilisée dans ce but est **ls** avec l'option **-l**. Revenons sur l'exemple d'utilisation de cette commande :

```
$ ls -l
total 2
-rw-r--r--  1 cpb      prof   18 sep  2 16:17 fic1
drwxr-x-w-  2 cpb      prof 1024 sep  2 16:18 un-rep
```


Nous savons que le propriétaire de *fic1* et *un-rep* est *cpb*, que leur groupe est *prof* et que les permissions sont **rw-r--r--** pour *fic1* et **rw-r-x-w-** pour *un-rep* (les 9 caractères qui suivent le premier, sur la ligne correspondante).

Les droits pour le propriétaire sont indiqués par les 3 premiers caractères des permissions, soit **rw-** pour *fic1* et **rw-x** pour *un-rep*. Ceux des membres du groupe du fichier sont indiqués par les 3 suivants (soit respectivement **r--** et **r-x**) et ceux des autres utilisateurs par les 3 derniers (**r--** et **-w-**).

Les droits sont eux-mêmes décomposables ainsi :

- 1<sup>er</sup> caractère : soit **r** (lecture permise), soit **-** (lecture interdite) ;
- 2<sup>e</sup> caractère : soit **w** (écriture permise), soit **-** (écriture interdite) ;
- 3<sup>e</sup> caractère : soit **x** (exécution permise), soit **-** (exécution interdite).

On déduit de l'exemple précédent que l'utilisateur *cpb* possède les droits de lecture et d'écriture sur *fic1*, et tous les droits (lecture, écriture et exécution) sur *un-rep*.

 Aucun droit sur un fichier n'est nécessaire pour en obtenir les détails (mais il faut le droit d'exécution sur le (chemin du) répertoire le contenant).

## 2.F.4 chmod : modifier les permissions de fichiers existants



**Il faut être très vigilant sur ce que l'on autorise, et à qui...**

Il arrive souvent que l'on veuille modifier manuellement les permissions accordées à un fichier, notamment pour interdire la lecture d'un fichier contenant des informations personnelles. Cette modification peut être opérée avec la commande externe **chmod**.

### Synopsis

**chmod** [-R] *permissions* *référence* { *référence* }

**chmod** fixe à *permissions*, les permissions sur les fichiers *référence*. L'option **-R** active le mode récursif pour appliquer les modifications sur l'arborescence de chaque *référence* répertoire, elle comprise. Les *permissions* peuvent être indiquées de deux manières : en utilisant le **mode absolu** ou le **mode symbolique**.



Aucune permission n'est nécessaire pour modifier les permissions d'un fichier (si ce n'est celles permettant de traverser le chemin qui y mène). Il faut juste en être le propriétaire (ou root).

### 2.F.4.a Mode absolu

Dans ce mode, *permissions* est un nombre (en octal) de la forme <sup>11</sup>  $C_u C_g C_o$  où  $C_u$ ,  $C_g$  et  $C_o$  sont 3 chiffres compris entre 0 et 7 :

- $C_u$  indique les permissions du propriétaire ;
- $C_g$  indique les permissions du groupe ;
- $C_o$  indique les permissions des autres.

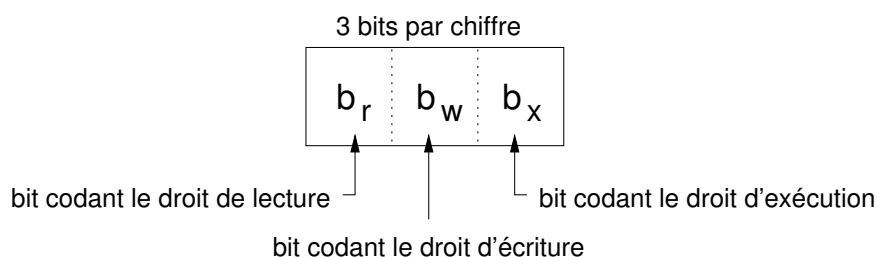
Chaque chiffre est obtenu en sommant des valeurs correspondant aux différents droits :

- 4 pour la lecture ;
- 2 pour l'écriture ;
- 1 pour l'exécution.

Un chiffre à 0 enlève tous les droits à la catégorie d'utilisateurs correspondante.



Plus exactement, chaque chiffre est un champ de 3 bits :



où chaque bit est à 1 si le droit correspondant est accordé, et à 0 sinon.

Ainsi, si un champ (chiffre) contient 110, c'est que la lecture et l'écriture sont accordées et pas l'exécution. On a bien  $110_2 = 100_2 + 010_2 = 4_{10} + 2_{10} = 6_{10}$ .

11. En réalité il y a un quatrième chiffre placé devant les 3 autres, mais nous n'en parlerons que plus tard.

## Exemples

```
$ ls -l
total 7
-rw-r--r--    1 cpb      prof      13 sep 10 02:15 fic1
-rw--w-r--    1 cpb      prof      13 sep 10 02:17 fic2
-rw-rw-rw-    1 cpb      prof      13 sep 10 02:18 fic3
-rwxr-xr-x    4 cpb      prof     4096 sep 08 05:21 rep
$ chmod 140 fic1
```

⇒ les permissions de *fic1* deviennent : exécution pour le propriétaire ( $C_u = 1$ ), lecture pour les membres du groupe ( $C_g = 4$ ), et rien pour les autres ( $C_o = 0$ ).

```
$ chmod 751 fic2 fic3
```

⇒ les permissions de *fic2* et *fic3* deviennent : tous les droits pour le propriétaire ( $C_u = 7$ ), lecture et exécution pour les membres du groupe ( $C_g = 5$ ), et exécution pour les autres ( $C_o = 1$ ).

```
$ chmod -R 700 rep
```

⇒ modifie les permissions de *rep* et de son aborescence.

```
$ ls -l
total 7
---xr-----    1 cpb      prof      13 sep 10 02:15 fic1
-rwxr-x--x     1 cpb      prof      13 sep 10 02:17 fic2
-rwxr-x--x     1 cpb      prof      13 sep 10 02:18 fic3
-rwx-----    4 cpb      prof     4096 sep 08 05:21 rep
```

□

### 2.F.4.b Mode symbolique

Dans ce mode, *permissions* a la forme suivante (les parenthèses ne servent ici qu'à la lisibilité) :

(*qui*) (*opérateur*) (*quoi*) { , (*qui*) (*opérateur*) (*quoi*) }



**La virgule est à écrire, et il n'y a pas d'espace.**

où *qui* indique à qui on veut fixer des permissions. C'est une combinaison des lettres **u**, **g**, **o** et **a**, représentant :

- **u** (*user*) le propriétaire ;
- **g** (*group*) les membres du groupe ;
- **o** (*others*) les autres ;
- **a** (*all*) à la fois le propriétaire, les membres du groupe et les autres.



*qui* peut être vide, ce qui correspond à **a** mais où les droits interdits par le masque (de **umask**) ne sont pas affectés par la modification.

(à méditer après avoir vu **umask**)

L'*opérateur* est l'un des signes **+**, **=** et **-**, pour indiquer si l'on veut :


- **+** ajouter,


- = fixer exactement,
- – supprimer

des permissions.

Enfin, *quoi* indique les permissions concernées par la modification. C'est une combinaison des lettres<sup>12</sup> **r**, **w**, **x** représentant :

- **r** (*read*) le droit de lecture ;
- **w** (*write*) le droit d'écriture ;
- **x** (*execute*) le droit d'exécution.

 *quoi* peut être vide, ce qui combiné avec =, permet d'enlever tous les droits aux individus indiqués par *qui*.

 Puisque **-r** est une opération valide, on comprend pourquoi l'option de récursivité est indiquée par **-R**.

Enfin, on peut écrire plusieurs opérations en les séparant par une virgule. Elles sont réalisées dans l'ordre comme le montrent les exemples qui suivent.

### Exemples

Reprenons l'exemple précédent mais utilisons le mode symbolique pour réaliser les mêmes changements de permissions.

```
$ ls -l
total 7
-rw-r--r--  1 cpb      prof      13 sep 10 02:15 fic1
-rw--w-r--  1 cpb      prof      13 sep 10 02:17 fic2
-rw-rw-rw-  1 cpb      prof      13 sep 10 02:18 fic3
-rwxr-xr-x  4 cpb      prof      4096 sep 08 05:21 rep
$ chmod u=x,o= fic1
```

⇒ *n'accorde que l'exécution pour le propriétaire (u=x), inchangé pour le groupe, enlève tous les droits aux autres (o=)*

```
$ ls -l fic1
---xr----- 1 cpb      prof      13 sep 10 02:15 fic1
```

⇒ *vérification des changements réalisés sur fic1*

```
$ chmod u+x,go=x,g+r fic2 fic3
```

⇒ *les opérations sont effectuées dans l'ordre : d'abord, ajoute l'exécution au propriétaire (u+x), puis n'accorde que l'exécution au groupe et aux autres (go=x) et enfin, ajoute la lecture au groupe (g+r)*

```
$ ls -l fic2 fic3
-rwxr-x--x  1 cpb      prof      13 sep 10 02:17 fic2
-rwxr-x--x  1 cpb      prof      13 sep 10 02:18 fic3
```

⇒ *vérification des changements réalisés sur fic2 et fic3*

```
$ chmod -R u=rwx,go= rep
$ ls -ld rep
-rwx----- 4 cpb      prof      4096 sep 08 05:21 rep
```

□

12. mais aussi **s** et **t** qui seront vus dans la section 14.B.3 page 203.

## 2.F.5 umask : enlever des permissions par défaut pour les nouveaux fichiers

Tous les fichiers ont leurs permissions fixées dès leur création. Elles dépendent de l'utilitaire employé et du type de fichier créé (voir ci-après), ainsi que du **masque de création de fichiers**. Ce dernier indique au système les permissions que ne doivent pas avoir les fichiers lors de leur création (uniquement). Sous Bash (et les principaux shells), il est consultable/modifiable par la commande interne **umask**.


### Synopsis

**umask** [*masque*]


Sans argument, **umask** indique la valeur actuelle du masque de création de fichiers, sinon *masque* est sa nouvelle valeur. L'argument *masque* est un nombre (en octal) de la forme  $C_u C_g C_o$  où  $C_u$ ,  $C_g$  et  $C_o$  sont 3 chiffres compris entre 0 et 7 :

- $C_u$  indique les permissions **à ne pas accorder** au propriétaire ;
- $C_g$  indique les permissions **à ne pas accorder** au groupe ;
- $C_o$  indique les permissions **à ne pas accorder** aux autres.

Chaque chiffre est obtenu en sommant les valeurs correspondant aux différents droits à exclure (4 pour la lecture, 2 pour l'écriture et 1 pour l'exécution). Un chiffre à 0 indique qu'aucun droit ne doit être enlevé. On peut remarquer que ces chiffres ont un rôle inverse de ceux de **chmod**.

 La modification du masque n'a aucune conséquence sur les fichiers existant ; le masque n'est utilisé que lors de la création de fichiers.

Le masque est propre à chaque processus (programme exécuté) mais se transmet entre processus père et fils (voir chapitres 10.B et 17). En conséquence, le masque d'un shell est transmis à tous les programmes exécutés depuis ce shell.


 La modification du masque dans un shell n'affecte pas les autres shells en cours d'exécution. Quand un shell est terminé, son masque disparaît avec lui.


### Précisions sur les permissions selon l'utilitaire et le type de fichier

Les utilisateurs créent leurs fichiers (au sens large) par le biais d'utilitaires (ce qui comprend le shell). Pour chaque type de fichier à créer, les utilitaires s'en remettent au système en faisant appel à une fonction appropriée de son API, et lui communiquent sa référence et son **mode**<sup>13</sup> (permissions). Si le système autorise la création du fichier, ses permissions seront le *mode* moins les permissions indiquées par le masque.

Les utilitaires n'ont *a priori* aucune raison de restreindre les permissions des fichiers qu'ils créent. Cela dit, la grande majorité des fichiers ordinaires contiennent du texte et des données, et n'ont pas vocation à être exécutés. C'est pourquoi, la plupart des utilitaires créent les fichiers ordinaires avec le mode 0666 (rw-rw-rw-). Il n'y a guère que les compilateurs pour en créer avec le mode 0777 (rwxrwxrwx). En créant une copie d'un fichier, **cp** fixe le même mode à la copie que celui du fichier source. Avec **mkdir**, les répertoires sont créés avec le mode 0777.

13. Pour les liens symboliques et les fichiers socket, le mode est fixé par le système.

 Vous exécutez probablement sans le savoir la commande **umask 002** (ou **umask 022** selon l'installation) à chaque fois que vous lancez un shell bash, car elle figure dans un fichier particulier qui est exécuté par le shell quand celui-ci démarre. Nous reviendrons sur ce type de fichier de démarrage dans la section 9.D page 107.


 Bash accepte l'expression d'un masque sous forme symbolique. Mais dans ce cas, on ne spécifie plus les droits qui doivent être enlevés mais ceux que l'on veut garder. Par ailleurs, les opérateurs **+** et **-** modifient le masque existant. Cette forme n'est pas utilisée dans ce document.

## Exemples


```
$ umask
```

```
0022
```

```
$ gedit fic1
```

 création d'un fichier *fic1* par l'éditeur **gedit**


```
$ mkdir rep1
```

 création d'un répertoire *rep1*


```
$ ls -l
```

```
total 8
```

```
-rw-r--r--    1 cpb      prof          13 sep 10 02:15 fic1
drwxr-xr-x    2 cpb      prof        1024 sep 10 02:07 rep1
```

 Conformément au masque, le droit d'écriture ne sera pas accordé au groupe ni aux autres. On remarque aussi que la permission d'exécution n'est pas non plus accordée (par **gedit**) aux fichiers ordinaires.

```
$ umask 267
```

 On ne veut pas que soient accordés le droit d'écriture pour le propriétaire, les droits de lecture et d'écriture pour le groupe et l'ensemble des droits pour les autres.


```
$ gedit fic2
```

```
$ mkdir rep2
```

```
$ ls -l
```

```
total 16
```

```
-rw-r--r--    1 cpb      prof          13 sep 10 02:15 fic1
-r-----    1 cpb      prof          13 sep 10 02:15 fic2
drwxr-xr-x    2 cpb      prof        1024 sep 10 02:07 rep1
dr-x--x---    2 cpb      prof        1024 sep 10 02:07 rep2
```

 les permissions de *fic1* et *rep1* n'ont pas changé.

```
$ umask 000
```

```
$ gedit fic3
```

```
$ mkdir rep3
```

```
$ ls -ld fic3 rep3
```

```
-rw-rw-rw-    1 cpb      prof          13 sep 10 02:18 fic3
drwxrwxrwx    2 cpb      prof        1024 sep 10 02:18 rep3
```

**i** Juste pour information, voici l’affichage du masque 000 sous forme symbolique en utilisant l’option **-S** de **umask** (non présente dans le synopsis de ce document) :

```
$ umask -S
u=rwx,g=rwx,o=rwx
```



## 2.F.6 chgrp : modifier le groupe d’un fichier

La commande externe **chgrp** permet au propriétaire d’un fichier ou à root d’en changer le groupe.

### Synopsis

```
chgrp [-fRv] groupe référence {référence}
```

Change le groupe des fichiers *référence* par le *groupe* indiqué. Le propriétaire d’un fichier peut en changer le groupe par un groupe dont il est membre. root peut le faire quels que soient le fichier et le groupe ciblé (qui doit exister).

L’option **-f** demande de ne pas afficher de message d’erreur si l’opération est impossible. L’option **-R** demande de faire ce changement récursivement sur les répertoires et leurs contenus. L’option **-v** active le mode verbeux (une ligne d’information par fichier modifié).

### Exemples

Supposons que l’utilisateur en session est toto, et qu’il appartient aux groupes users et etud1 :

```
$ ls -l fic1 fic2
-rw-r--r-- 1 toto etud1 13 sep 10 02:15 fic1
-rw-r--r-- 1 bob etud1 28 sep 12 10:21 fic2
$ chgrp users fic1
$ ls -l fic1
-rw-r--r-- 1 toto users 13 sep 10 02:15 fic1
```

⇒ toto a changé le groupe de son fichier

```
$ chgrp root fic1
chgrp: modification du groupe de « fic1 »: Opération non permise
```

⇒ mais il ne peut pas lui donner un groupe dont il n’est pas membre

```
$ chgrp users fic2
chgrp: modification du groupe de « fic2 »: Opération non permise
```

⇒ et ne peut pas non plus modifier le groupe d’un fichier qui ne lui appartient pas.

L’utilisateur root a plus de possibilités :

```
# chgrp root fic1
# ls -l fic1
-rw-r--r-- 1 toto root 13 sep 10 02:15 fic1
```

⇒ comme changer le groupe d’un fichier qui ne lui appartient pas...

```
# chgrp users fic2
# ls -l fic2
-rw-r--r--    1 bob      users      28 sep 12 10:21 fic2
```

⇒ ... et/ou donner à un fichier un groupe dont il n'est pas membre (mais qui doit exister).

□

## 2.F.7 chown : modifier le propriétaire et le groupe d'un fichier

La commande externe **chown** permet de modifier le propriétaire et le groupe d'un fichier ou d'une arborescence.

### Synopsis

```
chown [-fRv] utilisateur [: [groupe]] référence {référence}
```

Change le propriétaire des fichiers *référence*, qui devient *utilisateur* (qui doit exister). Si *utilisateur* est suivi d'un : (deux-points), alors change aussi le groupe : soit avec le *groupe* indiqué (qui doit exister), soit avec le groupe primaire de *utilisateur* si aucun *groupe* n'est spécifié.

L'option **-f** demande de ne pas afficher de message d'erreur si l'opération est impossible. L'option **-R** demande de faire ce changement récursivement sur les répertoires et leurs contenus. L'option **-v** active le mode verbeux.



**Seul root peut modifier le propriétaire d'un fichier.**

### Exemples

```
# ls -l fic1
-rw-r--r--    1 cpb      prof      13 sep 10 02:15 fic1
# chown toto fic1
# ls -l fic1
-rw-r--r--    1 toto     prof      13 sep 10 02:15 fic1
```

⇒ root n'a changé que le propriétaire du fichier

```
# chown bob: fic1
# ls -l fic1
-rw-r--r--    1 bob      users      13 sep 10 02:15 fic1
```

⇒ change à la fois le propriétaire et le groupe (par le groupe primaire de l'utilisateur). Ici, le groupe primaire de bob est users

```
# chown toto:etud1 fic1
# ls -l fic1
-rw-r--r--    1 toto     etud1      13 sep 10 02:15 fic1
```

⇒ change le propriétaire et le groupe (qui est spécifié).

□



# Chapitre 3

## Commandes : nature, localisation et aide

---

### 3.A Commandes internes, externes et le PATH

On peut distinguer deux types de commandes : les commandes internes et celles externes.

#### 3.A.1 Commandes externes et PATH

Les commandes externes sont les outils (utilitaires/applications) installés sur le système, fournissant des fichiers exécutables. Leur emplacement n'est pas imposé mais les outils standards utilisateur se trouvent dans des répertoires "classiques" tels que `/bin`, `/usr/bin` et `/usr/local/bin` (voir section 2.B). Ceux d'administration sont dans `/sbin`, `/usr/sbin` et `/usr/local/sbin`. Pour exécuter une commande externe, on peut toujours taper la référence absolue de son fichier exécutable.

#### Exemple

```
$ /bin/ls  
fic1  fic2  fic3  rep
```

⇒ exécute la commande externe `ls`, située dans `/bin`, pour afficher le contenu du répertoire de travail




**i** Un shell est une commande externe (comme `/bin/bash`). Une application de type terminal aussi (comme `/usr/bin/gnome-terminal`).

Utiliser une référence n'est pas pratique. Les shells permettent de simplifier cette écriture et d'invoquer les commandes externes simplement par le nom de leur fichier exécutable. Ils les recherchent alors dans une liste précise de répertoires, indiquée par la variable <sup>1</sup> d'environnement **PATH**. En principe, cette variable contient au moins les répertoires comme `/bin`, `/usr/bin` et `/usr/local/bin`.

---

1. voir sections 9.C (personnalisation du PATH), 15.A (variables) et 17.D (variables d'environnement).

 Pour connaître la liste des répertoire du PATH, il faut exécuter la commande :

**echo \$PATH**

Les répertoires de cette liste sont séparés par le caractère : (*deux-points*). L'ordre d'apparition des répertoires est important car les commandes sont recherchées dans cet ordre (de gauche à droite).

### Exemple

\$ **echo \$PATH**

/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games

⇒ les commandes externes sont recherchées dans les 5 répertoires mentionnés, en commençant par /usr/local/bin

\$ **ls**

fic1 fic2 fic3 rep

⇒ ls est trouvée dans /bin

\$ **inconnue**

-bash: inconnue : commande introuvable

⇒ inconnue n'existe dans aucun des répertoires du PATH

□

❗ Le **PATH** est personnalisable en modifiant la variable correspondante. Celui de root contient aussi les chemins vers les outils d'administration (/sbin, /usr/sbin et /usr/local/sbin).

## 3.A.2 Les commandes internes

À la différence des commandes externes qui sont "globales", les commandes internes (primitives du shell ou *builtins*) sont des commandes fournies par le shell lui-même. Chaque shell propose son propre jeu de commandes. Dans ce document, les commandes qualifiées d'internes sont celles de bash. Certaines sont indispensables<sup>2</sup> comme **cd** et **umask**. D'autres concernent des fonctionnalités de bash (**alias**, **shopt**, ...). Quelques unes sont des commandes d'usage courant qui existent déjà en tant que commandes externes (même nom, même fonction) comme **echo** et **pwd**. Elles sont peu nombreuses car elles sont embarquées dans le code de bash et occupent de la place en mémoire. Elles existent par ce que leur invocation est très rapide, ce qui accélère certaines tâches. Par exemple, pour **pwd**, il est bien plus efficace que bash affiche lui-même le répertoire de travail que de charger en mémoire l'utilitaire /bin/pwd pour si peu.


C'est pourquoi, bash donne la priorité à l'exécution de ses commandes internes (ce qui peut être modifiable). Elles ont parfois des différences avec les commandes externes, par exemple en n'admettant pas les mêmes options, ou en adoptant des comportements par défaut différents. C'est rarement gênant mais dans le cas où l'on a besoin d'utiliser la version externe, il suffit d'indiquer son chemin (ou de désactiver la commande interne avec **enable**).

2. Elles n'auraient pas de sens en tant que commandes externes.

### 3.A.3 Traitement du premier mot de la ligne de commandes

Sur une ligne de commandes, si le premier mot correspond à une commande interne, bash la reconnaît et l'exécute lui-même. Sinon, il doit s'agir<sup>3</sup> d'une commande externe qui correspond à un fichier exécutable. Pour trouver le fichier exécutable correspondant, bash suit le raisonnement suivant :

- si le mot contient un / (*slash*), alors celui-ci indique une référence relative ou absolue du fichier exécutable. Si la référence est correcte et le fichier est effectivement exécutable par l'utilisateur, il est exécuté sinon bash indique une erreur ;
- si le mot ne contient pas de *slash*, alors le fichier exécutable doit se trouver dans les chemins du **PATH**.

 Ainsi, si le répertoire de travail ne figure pas dans **PATH** et contient un fichier exécutable que l'on veut exécuter, on ne peut l'exécuter simplement en tapant son nom. Il faut préfixer le nom de `./` (ce qui donne une référence relative correcte et qui contient un *slash*).

#### Exemples

Supposons que le répertoire de travail (ici *rep*) ne figure pas dans le PATH et contienne un fichier exécutable appelé *utilitaire* :


```
rep$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

 le PATH contient 5 répertoires

```
rep$ ls -l utilitaire
-rwxr--r-- 1 toto toto 28419 20 janv. 2010 utilitaire
```


 la commande externe **ls** est trouvée dans */bin*. Le fichier *utilitaire* est bien exécutable par *toto*

```
rep$ utilitaire
-bash: utilitaire: command not found
```

 puisque le premier mot ne contient pas de /, bash recherche la commande dans les répertoires du PATH et ne la trouve pas


```
rep$ ./utilitaire
```

... exécution de utilitaire ...

 le premier mot contenant un /, la commande n'est pas recherchée dans le PATH et puisque la référence est correcte, le fichier est exécuté

```
rep$ ../rep/utilitaire
```

... exécution de utilitaire ...

 à nouveau, le premier mot contenant un /, la commande n'est pas recherchée dans le PATH et puisque la référence est correcte, le fichier est exécuté

□

3. Ce peut être aussi un alias, une fonction ou un mot-clé de bash, ce que nous verrons plus tard.

## 3.B Identifier les commandes

### 3.B.1 type : obtenir la nature d'une commande

La commande interne **type** demande à bash d'indiquer la nature des mots qui lui sont passés en arguments. Elle permet notamment de savoir si une commande est interne ou externe.

#### Synopsis

```
type [-afptP] mot {mot}
```

Indique pour chaque *mot*, comment il sera interprété par bash. Il peut s'agir dans l'ordre :

- d'un alias (nous les verrons dans la section 9.A page 101) ;
- d'un mot-clé de bash comme **function**, **if**, **for**, etc. (nous les verrons plus tard) ;
- d'une fonction (nous les verrons dans la section 9.B page 104) ;
- d'une commande interne (ou *shell builtin* ou primitive du shell) ;
- d'une commande externe (un fichier exécutable) ;
- d'un mot inconnu.

Sans option, indique la première correspondance de *mot* dans la liste ci-dessus. S'il s'agit d'une fonction, sa description est aussi affichée. L'option **-a** demande toutes les correspondances (si *mot* en a plusieurs : par exemple, existe en tant que commande interne mais aussi comme commande externe). L'option **-f** demande de ne pas rechercher parmi les fonctions. L'option **-p** demande d'afficher la référence du fichier à exécuter si le mot est uniquement une commande externe (figurant dans un chemin du path), et rien sinon. L'option **-P** agit comme **-p** mais limite la recherche aux commandes externes. L'option **-t** demande simplement l'affichage du type de mot (*alias*, *keyword*, *function*, *builtin* ou *file*).

#### Exemples

```
$ type help
```

```
help is a shell builtin
```

⇒ *help est une commande interne*

```
$ type man
```

```
man is /usr/bin/man
```

⇒ *alors que man est externe (le fichier exécutable /usr/bin/man)*

```
$ type echo
```

```
echo is a shell builtin
```

⇒ *echo est une commande interne*

```
$ type -a echo
```

```
echo is a shell builtin
```

```
echo is /bin/echo
```

⇒ *echo existe en deux versions : interne et externe. C'est la version interne qui est exécutée par défaut.*

```
$ type -t echo
```

```
builtin
```

```
$ type -p echo
```

⇒ *n'affiche rien car echo est aussi une commande interne*

```
$ type -P echo
/bin/echo
```

⇒ on obtient le chemin de la commande externe

```
$ type trucmuche
-bash: type: trucmuche: not found
```

□

### 3.B.2 which : afficher le chemin complet d'une commande externe

La commande **which** recherche et affiche le chemin (référence) complet d'une commande (externe).

#### Synopsis

```
which [-a] commande {commande}
```

Pour chaque *commande* en argument, **which** cherche et affiche la référence du fichier exécutable correspondant. La recherche est la même que celle opérée par les shells pour exécuter une commande externe. Elle est basée sur le contenu de la variable **PATH** en recherchant dans les répertoires qu'elle contient, et dans leur ordre d'apparition. Par défaut, **which** n'affiche que la première référence trouvée et n'en recherche pas d'autres dans la suite du PATH. L'option **-a** (*all*) demande de continuer et de les afficher toutes.

✍ **which** est indépendante du shell et repose uniquement sur le PATH. Elle ne tient donc pas compte des commandes internes du shell courant, des alias, fonctions ou autres. Pour savoir effectivement comment un shell exécute une commande, il est préférable d'utiliser sa commande interne dédiée. Dans le cas de bash, il s'agit de **type** (voir section 3.B.1 page précédente). Dans le cas de (t)csh, il s'agit de **which** !

#### Exemples

```
$ echo $PATH
/home/marcel/bin:/usr/local/bin:/usr/bin:/bin:/usr/games:/sbin:/usr/sbin:.
$ which echo
/bin/echo
```

⇒ */bin* est le premier répertoire du PATH contenant le fichier exécutable **echo**, dont **which** affiche la référence

```
$ cp /bin/echo ~/bin
```

⇒ copie de l'exécutable */bin/echo* dans le répertoire */home/marcel/bin* situé avant */bin* dans le PATH.

```
$ which echo
/home/marcel/bin/echo
```

⇒ c'est le fichier qui serait maintenant exécuté pour **echo**

```
$ which -a echo
/home/marcel/bin/echo
/bin/echo
```

⇒ affiche toutes les correspondances du PATH

```
$ which ls kill
/bin/ls
/bin/kill
```

⇒ recherche plusieurs commandes

```
$ type echo ls kill
echo est une primitive du shell
ls est un alias vers « ls --color=auto »
kill est une primitive du shell
```

⇒ Néanmoins, **which** ne tient pas compte des éléments internes du shell. Notamment, **echo** et **kill** sont en réalité des commandes internes de **bash**, exécutées à la place de `/bin/echo` et `/bin/kill`, sauf dispositions contraires.

□

### 3.B.3 whereis : rechercher les fichiers relatifs à une commande externe

La commande **whereis** permet de rechercher les exécutables, les sources et les pages de manuel d'une commande.

#### Synopsis

```
whereis [-bms] [-BMS répertoire {répertoire} -f] fichier {fichier}
```

Sans option, **whereis** recherche les exécutables, sources et pages de manuel relatives à chaque *fichier* en arguments. Généralement, *fichier* correspond au nom d'une commande. Pour plus de souplesse, il peut contenir un chemin et/ou une extension, auquel cas **whereis** ne retiendra que le nom du fichier sans extension.

#### Exemples

```
$ whereis ls
ls: /bin/ls /usr/share/man/man1/ls.1.gz
$ whereis /par/ici/ls.c
ls: /bin/ls /usr/share/man/man1/ls.1.gz
```

□

L'option **-b** demande de rechercher les exécutables. L'option **-m** demande de rechercher les pages de manuel. L'option **-s** demande de rechercher les fichiers sources. Le défaut est la composition de ces 3 options.

La recherche des différents fichiers se fait par défaut dans les répertoires standards, tels que `/bin`, `/usr/man`, `/usr/src`, etc. L'option **-B** indique les *répertoires* où chercher les exécutables. L'option **-M** indique les *répertoires* où chercher les pages de manuel. L'option **-S** indique les *répertoires* où chercher les sources. Ces options doivent être suivies de **-f** qui sépare les *répertoires* des *fichiers* à rechercher.

#### Exemples

```
$ whereis -b ls
ls: /bin/ls
$ whereis -m ls
ls: /usr/share/man/man1/ls.1.gz
$ whereis -s ls
ls:
```

□

## 3.C Aide en ligne

Ainsi qu'il a été indiqué dans l'avant-propos, la description des commandes dans ce document n'est que partielle. Je n'ai indiqué que les options qui me semblent les plus utiles. Il n'est pas possible de les décrire toutes de manière exhaustive, d'autant que les utilitaires évoluent et que de nouvelles options apparaissent régulièrement. Selon vos besoins, il faudra donc chercher de l'information ailleurs, et notamment dans l'aide en ligne.

### 3.C.1 help : aide sur les commandes internes de bash


Bash ne fait pas qu'interpréter des lignes de commandes. Il fournit lui-même un certain nombre de commandes que l'utilisateur peut lui demander d'exécuter. Ces commandes sont chargées en mémoire lorsqu'on exécute bash et sont donc rapidement exécutées. On les appelle les **commandes internes**. L'utilisateur peut aussi exécuter d'autres commandes non fournies par bash. Ce sont les **commandes externes**. Elles correspondent à des fichiers exécutables présents sur le système. Dans ce cas, le fichier correspondant doit être chargé en mémoire pour pouvoir être exécuté, ce qui prend du temps. À des fins d'optimisation, certaines commandes externes ont été intégrées dans bash en tant que commandes internes, parfois avec des options différentes. Par exemple, la commande **echo** est à la fois une commande interne de bash, mais correspond aussi à un fichier exécutable (commande externe). Si un utilisateur veut exécuter **echo**, c'est la commande interne qui sera préférée. Mais, il sera toujours possible pour l'utilisateur de forcer bash à utiliser la version externe.

La commande **interne help** demande l'affichage d'une aide **sur des commandes internes** passées en arguments.

#### Synopsis

```
help [-s] {commande}
```

Sans option ni argument, affiche la liste des commandes internes de bash avec leurs options. Sinon, pour chaque *commande*, affiche une description de la commande et son synopsis. *commande* peut être un nom partiel auquel cas toutes les commandes commençant par ce nom seront affichées. L'option **-s** demande de ne pas afficher l'aide complète sur *commande* mais simplement son synopsis.

 *commande* peut aussi être un **motif** (nous verrons les motifs dans la section 4.D page 56).

#### Exemples

```
$ help
GNU bash, version 3.1.17(1)-release (i486-pc-linux-gnu)
These shell commands are defined internally.  Type 'help' to see this list.
Type 'help name' to find out more about the function 'name'.
Use 'info bash' to find out more about the shell in general.
Use 'man -k' or 'info' to find out more about commands not in this list.
```

A star (\*) next to a name means that the command is disabled.

```
JOB_SPEC [&]                (( expression ))
. filename [arguments]      :
[ arg... ]                  [[ expression ]]
alias [-p] [name[=value] ...] bg [job_spec ...]
bind [-lpvsPVS] [-m keymap] [-f fi break [n]
builtin [shell-builtin [arg ...]] caller [EXPR]
```

```

case WORD in [PATTERN [| PATTERN]. cd [-L|-P] [dir]
command [-pVv] command [arg ...] compgen [-abdefgjkusv] [-o option]
complete [-abdefgjkusv] [-pr] [-o continue [n]]
declare [-afFirtx] [-p] [name[=val] dirs [-clpv] [+N] [-N]]
disown [-h] [-ar] [jobspec ...] echo [-neE] [arg ...]
... ..

```

⇒ affiche la liste (ici tronquée) des commandes internes et leurs options.

✍ Dans ce qu’écrit **help**, les formes `[arg ...]` ou `[jobspec ...]` ont le même sens que les formes `{arg}` et `{jobspec}` dans notre convention. C’est aussi le cas pour les pages de manuel (voir ci-après).

\$ **help help cd**

help: help [-s] [pattern ...]

Display helpful information about builtin commands. If PATTERN is specified, gives detailed help on all commands matching PATTERN, otherwise a list of the builtins is printed. The -s option restricts the output for each builtin command matching PATTERN to a short usage synopsis.

cd: cd [-L|-P] [dir]

Change the current directory to DIR. The variable \$HOME is the default DIR. The variable CDPATH defines the search path for the directory containing DIR. Alternative directory names in CDPATH are separated by a colon (:). A null directory name is the same as the current directory, i.e. `.`. If DIR begins with a slash (/), then CDPATH is not used. If the directory is not found, and the shell option `cdable\_vars' is set, then try the word as a variable name. If that variable has a value, then cd to the value of that variable. The -P option says to use the physical directory structure instead of following symbolic links; the -L option forces symbolic links to be followed.

⇒ aide sur les commandes internes **help** et **cd**

\$ **help -s help**

help: help [-s] [pattern ...]

⇒ synopsis de la commande **help**

□

### 3.C.2 man : le manuel en ligne

Pour les commandes **externes**, on dispose de la commande **man**. Tout informaticien travaillant sous Unix, l’utilise régulièrement, à part peut-être ceux qui ont une mémoire phénoménale. C’est véritablement une mine d’informations, à consulter sans modération.

#### Synopsis

**man** [section] [-k expreg {expreg}] | -f mot {mot} | [-t] commande]

**man** permet d’obtenir à l’écran la documentation officielle (de plus en plus souvent en français) de la *commande* passée en argument. Cette documentation est organisée en au moins 9 sections regroupant chacune un type de documentation. Ces sections sont les suivantes :



Section	Type de contenu
1	Commandes utilisateur
2	Appels système (fonctions fournies par le kernel, prototypées en langage C)
3	Fonctions fournies par des bibliothèques
4	Fichiers spéciaux, périphériques (en général placés dans le répertoire <code>/dev</code> )
5	Formats de fichiers particuliers et conventions (comme <code>/etc/passwd</code> )
6	Jeux
7	Divers
8	Commandes d'administration système
9	Section non standardisée, au contenu variable
n	Nouveautés (par exemple, commandes du langage Tcl/Tk)

Si *section* n'est pas mentionnée, **man** recherche la documentation de commande dans l'ordre croissant des sections et affiche la première documentation trouvée. Sinon, *section* doit être un chiffre de **1** à **9** ou **n**.

L'affichage de la documentation est paginé à l'écran (affiché page par page). Lorsqu'une page est affichée, l'utilisateur peut choisir :

- d'arrêter la commande (et l'affichage) en tapant sur la touche `q` ;
- de faire défiler la page d'une ligne en tapant sur `Entrée` ;
- de faire afficher la page suivante en tapant sur la barre d'espaces ;
- de faire afficher la page précédente en tapant sur `b` ;
- et bien d'autres choses encore...

L'option **-k** est suivie d'une expression régulière (voir chapitre 12 page 151) et donne le même résultat que la commande **apropos** (voir ci-après).

L'option **-f** est suivie d'un mot et donne le même résultat que la commande **whatis**.

L'option **-t** demande à ce que la documentation soit écrite sur la sortie standard<sup>4</sup> mais au format *PostScript*<sup>5</sup> pour qu'elle soit facilement imprimable.

## Exemples

```
$ man kill
```

⇒ affiche la documentation sur la commande **kill** fournie par le fichier exécutable `/bin/kill` (et non pas la commande interne **kill** de `bash`) ;

```
$ man 2 kill
```

⇒ affiche la documentation de la **fonction** du langage C `kill()`.

□

 Il est même possible de consulter la documentation sur la commande **man** en tapant **man man**.

4. Nous reviendrons sur cette notion dans le chapitre 5 page 61.

5. Le format *PostScript* est un format de mise en forme de texte inventé par la société Adobe (qui a aussi inventé le format PDF), et reconnu par de nombreuses imprimantes actuelles, en particulier celles du Département Informatique.

❗ **man** est indispensable sur un système Unix, mais reste assez basique. Pour amener plus de convivialité, notamment permettre une navigation dans des rubriques, la commande **info** a été développée et son utilisation tend à se généraliser. Il est désormais préférable d'utiliser **info** si elle est disponible car les documentations sont plus détaillées que celles de **man**. À noter que si **info** ne trouve pas la documentation d'une commande dans son format (permettant la navigation), alors c'est le format **man** qui sera affiché.

### 3.C.3 whatis/apropos : rechercher des pages de manuel

Les commandes **whatis** et **apropos** permettent de rechercher les pages de manuel qui concernent des chaînes.

#### Synopsis

```
whatis mot {mot}
apropos expreg {expreg}
```

**whatis** affiche une ligne par page du manuel qui possède *mot* comme mot-clef. **apropos** a la même fonction mais admet une expression régulière (voir chapitre 12 page 151) plutôt qu'un mot.

❗ Pour que ces commandes fonctionnent, il faut que la base de données *whatis* ait été créée par root par la commande **makewhatis**.

#### Exemples

```
$ whatis lpr
lpr: nothing appropriate
$ apropos lpr
lpr: nothing appropriate
```

⇒ visiblement la base *whatis* n'est pas créée (car **lpr** est une commande qui existe et est documentée)

```
# /usr/sbin/makewhatis
```

⇒ demande la création de la base *whatis* (en tant que root)

```
$ whatis lpr
lpr                (1)  - Imprime des fichiers
lpr                (1)  - print files
lpr [lpr-cups]     (1)  - print files
⇒ cette fois les entrées où apparaît le mot lpr sont affichées
```

```
$ apropos lpr
glPrioritizeTextures (3x) - set texture residence priority
lpdomatic            (8)  - Foomatic filter script for LPD, LPRng, and GNUlpr
lpr                  (1)  - Imprime des fichiers
lpr [lpr-cups]       (1)  - print files
lprm                 (1)  - Annule des travaux d'impression
lprm [lprm-cups]     (1)  - cancel print jobs
QSqlPropertyMap [qsqlpropertymap] (3qt) - Used to map widgets to SQL fields
⇒ les entrées où apparaît la chaîne lpr sont affichées
```

□

# Chapitre 4

## Bash : composition d'une ligne de commandes


---

### 4.A Introduction

Dans le chapitre 1, nous avons évoqué le découpage de la ligne de commandes en mots à l'aide de **blancs** (espace et tabulation), notamment afin de déterminer la commande à exécuter et ses paramètres (options et arguments). Les blancs jouent donc le rôle de **séparateurs de mots** mais ce ne sont pas les seuls. Il y a aussi `|` (tube ou *pipe*), `&` (esperluette ou et-commercial), `;`, `(`, `)`, `<`, et `>`. Les séparateurs de mots sont appelés des **méta-caractères**. Nous verrons au fil du document comment les méta-caractères permettent d'agir sur le contexte d'exécution des commandes.

Outre les méta-caractères, bash reconnaît des (combinaisons de) caractères spéciaux qu'il traite de manière spéciale. Ce traitement permet notamment d'alléger/faciliter l'écriture des lignes de commandes. En effet, avant de les exécuter réellement, bash transforme les lignes de commandes qui comportent certains caractères spéciaux. Dans l'ordre, ces transformations sont :

- expansion des accolades
- expansion du tilde
- expansion des paramètres et des variables (voir chapitre 15)
- substitution de commandes (voir chapitre 18)
- expansion des expressions arithmétiques (voir chapitre 18)
- décomposition en mots (voir chapitre 18)
- expansion des chemins (ou noms) de fichiers, appelée aussi substitution des motifs de noms de fichiers.

 Dans ce document, le terme *substitution* est parfois employé à la place d'expansion.

Pour donner un aperçu de la facilité d'écriture due à ces transformations, considérons l'expansion des chemins de fichiers. Elle permet de référencer très simplement un ensemble de fichiers, en utilisant des **caractères spéciaux** dans les références, constituant ainsi un motif de chemin de fichier qui sera remplacé par la liste des fichiers qui correspondent au motif. Par exemple, le motif `*.txt` sera remplacé par le nom de tous les fichiers du répertoire de travail, portant l'extension `.txt`. L'utilisateur peut taper ce motif à la place des fichiers correspondants.

Ce chapitre présente les expansions des accolades, du tilde et des chemins de fichiers. D'abord, passons en revue les caractères spéciaux.

## 4.B Les caractères spéciaux de bash

Les caractères spéciaux (y compris les **méta-caractères**), ont une signification particulière selon leur position lorsqu'ils se trouvent sur la ligne de commande. Ils sont interprétés par bash conformément à leur signification. Les méta-caractères séparent les mots<sup>1</sup> mais ont une autre signification, à part l'espace et la tabulation dont c'est l'unique rôle. Par exemple, le méta-caractère `;` sépare aussi les commandes. Le caractère de retour à la ligne (obtenu au clavier avec la touche Entrée) demande l'interprétation de la ligne de commandes. Il sépare donc aussi les commandes entre deux lignes lues.

Les caractères spéciaux pour bash sont : retour à la ligne, `\` (barre oblique inversée ou *backslash*), `~` (tilde), `#`, `$`, `'` (apostrophe ou *quote*), ``` (apostrophe inversée ou *backquote*), `*` (étoile), `?`, `!`, `{`, `}`, `"`, `[`, `]`, ainsi que les méta-caractères (espace, tabulation, `|`, `&`, `;`, `(`, `)`, `<`, `>`).

Le `\` (*backslash* appelé aussi *anti-slash* ou contre-oblique) sert à protéger le caractère qui le suit, de façon à ce qu'il **garde son sens littéral**, c'est à dire que bash ne l'interprète pas comme un caractère spécial s'il en est un.

### Exemple

Supposons que dans le répertoire de travail, un nom de fichier contienne un espace tel que `un_fichier.txt`. Si l'on veut en faire une copie nommée `fic`, il est incorrect de taper (en faisant apparaître les espaces) :


```
$ cp_un_fichier.txt_fic
```

car cela demande de copier les deux fichiers `un` et `fichier.txt`. Il faut protéger l'espace du nom ce qui donne :

```
$ cp_un\_fichier.txt_fic
```

□

Avant d'exécuter une ligne de commandes, bash l'analyse pour déterminer s'il s'agit d'une ou plusieurs commandes, leurs paramètres (arguments et options) et d'autres choses. Pour cela, il se base sur les caractères spéciaux non protégés et prépare la (ou les) commandes à exécuter en fonction de leur signification. Les caractères spéciaux non protégés sont supprimés de la ligne de commande après avoir été traités par le shell. Dans l'exemple précédent, le *backslash* qui protégeait l'espace est supprimé par bash, il ne reste que les deux arguments de `cp` : `un_fichier.txt` et `fic`.

 Pour utiliser des caractères spéciaux en argument d'une commande pour laquelle ils ont une signification, il faut les protéger avec le *backslash* pour ne pas que le shell les interprète. On verra une autre méthode de protection dans la section 4.F page 59.

Notons que certains caractères spéciaux deviennent normaux s'ils sont encadrés par certains autres caractères spéciaux. Quelques rares caractères normaux (non spéciaux) deviennent spéciaux s'ils sont encadrés par certains caractères spéciaux. Notons aussi que certains caractères spéciaux peuvent se composer, ce qui a encore une signification spéciale comme : `<<`, `>>`, `&&`, `|`, ...

1. Pendant l'étape de décomposition en mots, qui se produit dans certaines conditions, bash utilise la variable `IFS` pour déterminer les séparateurs de mots. Cette variable contient par défaut la chaîne `'\t\n'`, représentant les caractères espace, tabulation et retour à la ligne.

❗ Seuls la pratique et des tests permettent de se familiariser avec ces caractères et leur rôle... Nous verrons leur utilisation tout au long de ce cours.

## 4.C Expansion du tilde

Si un mot de la ligne de commandes commence par un tilde (~) non protégé, alors tous les caractères de ce mot qui suivent ce tilde, jusqu'au premier slash non protégé forment un **tilde-préfixe**. Si aucun caractère de ce tilde-préfixe n'est protégé, le tilde-préfixe est traité comme un éventuel nom d'utilisateur. Si cet utilisateur existe, alors le tilde et le tilde-préfixe sont remplacés par le chemin absolu du répertoire d'accueil de cet utilisateur. Si aucun utilisateur de ce nom n'existe, le mot est laissé inchangé. Enfin, si le tilde-préfixe est vide, le tilde est remplacé par le chemin absolu du répertoire d'accueil<sup>2</sup> de l'utilisateur en session (c'est à dire qui exécute bash).

❗ Nous avons déjà vu le tilde lorsque nous avons présenté l'invite de commande, section 1.F.2 page 6, où nous avons indiqué que le tilde seul représentait le répertoire d'accueil de l'utilisateur en session.

### Exemples

Supposons que l'utilisateur en session soit toto, et que l'utilisateur bob existe mais qu'il n'existe pas d'utilisateur titi, alors :

```
toto@allegro:~/rep$ ls ~bob
```

⇒ le tilde-préfixe est bob. La commande affiche, si cela est permis, le contenu du répertoire d'accueil de bob, car ~bob est remplacé par /home/bob

```
toto@allegro:~/rep$ cp ~bob/fic ~
```

⇒ le tilde-préfixe est encore bob, car il s'arrête au premier slash non protégé. La commande tente de copier le fichier fic du répertoire d'accueil de bob dans le répertoire d'accueil de toto

```
toto@allegro:~/rep$ cp ~b\ob/fic .
```

⇒ ici, un caractère du tilde-préfixe est protégé, l'expansion n'a donc pas lieu. Au final, on veut copier le fichier fic du répertoire ~bob (du répertoire de travail) dans le répertoire de travail

```
toto@allegro:~/rep$ cp ~titi/fic .
```

⇒ de même, l'utilisateur titi n'existant pas, il n'y a pas d'expansion. Au final, on veut copier le fichier fic du répertoire ~titi (du répertoire de travail) dans le répertoire de travail

□

✍ Dans le cas de l'affectation à une variable (étudiée en détail au chapitre 15 page 207), l'expansion du tilde peut avoir lieu si la tilde est placée après le = ou après un : (deux points).

2. En réalité, le tilde seul est remplacé par le contenu de la variable HOME.

## 4.D Expansion des chemins de fichiers

**i** Les **motifs** sont des descriptions incomplètes de références de fichiers (au sens large) et **sont remplacés par la liste** (en principe) **triée des références qui correspondent** à cette description.


En fait, toute référence peut être vue comme un motif même celles ne comportant pas de caractères spéciaux. En effet, un caractère non spécial apparaissant dans une référence ne signifie rien d'autre que lui-même.

Les motifs peuvent apparaître dans n'importe quel mot de la ligne de commandes. Un motif (ou une référence) est une succession de caractères normaux (non spéciaux) et de caractères spéciaux. Les caractères spéciaux pour les motifs de noms de fichiers sont : `?`, `*` et `[`, `]`, auxquels on peut ajouter `^`, qui n'a une signification spéciale<sup>3</sup> que placé entre crochets, immédiatement après le crochet ouvrant. Lorsque le shell rencontre ces caractères non protégés, alors la chaîne correspondante est candidate à l'expansion des chemins de fichiers, appelée aussi « développement » (*globbing*).

La signification de ces caractères est décrite ci-dessous :

- `?` : représente n'importe quel caractère sauf le slash et le point<sup>4</sup> qui cache un fichier (celui qui commence son nom) ;
- `*` : représente n'importe quelle suite de caractères dépourvue de slash et du point<sup>4</sup> qui cache un fichier, même la suite vide ;
- `[ [^] ensemble ]` : représente un caractère appartenant à un ensemble de caractères qui ne comprend pas de slash. Le caractère spécial `[` marque le début de la définition d'un ensemble de caractères. Celle-ci doit être terminée par le caractère spécial `]`. L'ensemble représenté entre crochets dépend de la présence ou non du caractère `^` après le `[`. S'il n'est pas présent, c'est l'ensemble défini par *ensemble*. Sinon, c'est son complément (c.-à-d. tout caractère autre que celui de *ensemble*). *ensemble* est composé de caractères normaux collés les uns aux autres mais le caractère `-` (tiret) a une signification particulière : `c1-c2` (où `c1` et `c2` sont deux caractères normaux) représentent tous les caractères de `c1` à `c2`. Par exemple, `a-e` représente l'ensemble `a`, `b`, `c`, `d`, `e`. En revanche, si le tiret est placé en début ou en fin d'ensemble, alors il ne représente que lui-même.

**i** À noter que le caractère `!` joue le même rôle que `^`.

 Selon comment bash a été installé (et compilé), l'ordre utilisé pour classer les caractères peut être différent que le simple ordre ASCII. La tendance est à mélanger les majuscules, les minuscules et les caractères accentués, faisant que `a`, `A` et `à` sont "plus petits" que `B`, ce qui n'est pas le cas en ordre ASCII. Cela influe notamment sur le traitement des ensembles, sur la liste des fichiers correspondant à un motif et sur son classement. Pour forcer le classement des caractères selon leur codage ASCII, il faut modifier la variable `LC_COLLATE` en exécutant :

```
$ LC_COLLATE=C
```

Cela n'impacte que le shell courant mais on peut placer cette commande dans un fichier de configuration de bash (voir chapitre 9).

3. En réalité, `^` est aussi utilisé pour le développement de l'historique.

4. À moins que l'option `dotglob` de bash soit activée.

❗ Les caractères `?`, `*`, et `[` perdent leur signification à l'intérieur de la définition d'un ensemble (c'est-à-dire à l'intérieur des crochets).

💣 Les blancs ne sont pas protégés entre crochets.

❗ Le caractère `]` peut aussi figurer dans la définition de *ensemble* à condition qu'il en soit le premier caractère.

### Exemple

En se basant sur l'architecture présente sur *allegro*, la commande :

```
$ cp ~toto/*.txt .
```

copie dans le répertoire de travail tous les fichiers se terminant par `.txt` contenus dans le répertoire d'accueil de l'utilisateur *toto*.

🔗 L'interpréteur de commande a transformé le tilde et le motif avant même de lancer la commande. Puis, si la liste obtenue n'est pas vide, celle-ci remplace le motif dans la ligne de commandes qui est lancée. Si aucun fichier ne correspond, *bash* ne produit pas d'erreur et le motif est considéré comme un mot normal. Cependant, en activant l'option *failglob* (avec la commande **shopt**), un motif sans correspondance provoque une erreur de *bash*.

□

### Exercice

Si le répertoire d'accueil de *toto* se trouve dans `/home`, ne contient que les répertoires `.`, `..`, `tp` et les fichiers `un_fichier.txt`, `another.fic.txt`, `un_outil` et `prog.ads`, quelle est la ligne de commandes qui est finalement exécutée ?

### Exercice

Donnez les motifs qui représentent les énoncés suivants :

1. liste (ordonnée) des fichiers du répertoire courant commençant par la lettre **a** ou **A** et terminée par n'importe quoi ?
2. liste des fichiers qui se terminent par l'extension **.cxx** du répertoire `tp` contenu dans le répertoire d'accueil de *toto* ?
3. liste des fichiers du répertoire courant qui commencent par un caractère quelconque, suivie d'une lettre minuscule ou majuscule, suivie de n'importe quoi et qui se terminent par **.ad** suivi soit de **b** soit de **s** ?
4. liste des fichiers du répertoire courant dont le nom contient une étoile ?
5. liste des fichiers contenus dans le répertoire d'accueil de tous les utilisateurs ?
6. liste des fichiers cachés contenus dans le répertoire d'accueil de tous les utilisateurs ?

### Corrigé

1. `[aA]*`
2. `~toto/tp/*.cxx`
3. `?[a-zA-Z]*.ad[bs]`
4. `*[*]*` mais aussi `*\**`  
 ➡ et on verra encore `*' *' *` ou même `*" *" *`
5. `/home/*/*`  
 ➡ les fichiers cachés ne correspondent pas car commencent par un `.`
6. `/home/*/. *`  
 ➡ Seuls les fichiers cachés comme `., ., ., .bash_profile` correspondent.

**i** Il est possible de développer le motif que l'on tape (c.-à-d. le remplacer par les fichiers qu'il représente), avant de lancer la commande. Pour cela, le faut placer le curseur à la fin du motif et taper la combinaison de touches `CTRL-X *` qui a été liée (par la commande interne **bind**) à la commande d'édition de bash *glob-expand-word*. On peut alors modifier cette liste. Pour simplement afficher les fichiers correspondant au motif courant, on tapera `CTRL-X g` correspondant à la commande d'édition *glob-list-expansions*.

## 4.E Expansion des accolades

Les accolades sont un cas à part car, contrairement aux expansions précédentes, elle ne dépendent pas des fichiers ou utilisateurs existants. Elles ne sont interprétées que si une virgule apparaît à l'intérieur, ou dans certains cas `..` (deux points consécutifs).

Lorsque bash rencontre une construction de type :

$$X\{motif_1, motif_2, \dots, motif_n\}Y$$

où  $X$  et  $Y$  sont aussi des motifs, alors bash remplace cette liste par :

$$Xmotif_1Y \_ Xmotif_2Y \_ \dots \_ Xmotif_nY$$

Les différents motifs peuvent être vides.

Ce développement laisse les motifs dans l'ordre, bien que chaque motif puisse lui-même être remplacé par une liste triée de noms de fichiers. Par exemple, le motif `{truc,bidule}.txt` est remplacé par `truc.txt bidule.txt`. Le motif `{tr*,bidule}.txt` pourrait lui être remplacé par `trac.txt truculent.txt bidule.txt` (les noms `trac.txt` et `truculent.txt` ont été triés).

La présence d'accolades a plusieurs conséquences qui viennent quelque peu contredire ce qui a été dit sur les motifs :

- la liste de noms obtenue n'est plus forcément triée ;
- lorsque le motif est développé, les noms obtenus ne correspondent plus forcément à des fichiers existants.

### Exemples



1. La liste des fichiers des répertoires */bin* et */usr/bin* est représentée par :

`{/usr/,/}bin/*`

ou bien :

`/ {usr/, }bin/*`

ou encore :

`{/usr, }/bin/*`

2. La liste des fichiers des utilisateurs bob et toto qui sont contenus dans le répertoire *public* et qui se terminent par *.c*, *.cxx* ou *.java* est représentée par

`~{bob,toto}/public/*. {c,c++,java}`

ou encore :

`~{bob,toto}/public/*. {c{,++},java}`

□

❗ Dans les accolades, on peut utiliser à la place des motifs la construction  $c_1 . . c_2$ , où  $c_1$  et  $c_2$  sont deux caractères, qui est remplacée par la liste des caractères entre  $c_1$  et  $c_2$  (eux compris). Mais au contraire de la transformation précédente, celle-ci n'a pas lieu s'il y a une virgule.

## Exemples

1. `{a..e}` est remplacé par *a b c d e*
2. `{e..a}` est remplacé par *e d c b a*
3. `{a..e, fic}*` est remplacé par *a..e\* fic\** qui seront aussi soumis à l'expansion des chemins de fichiers (ce sont tous des motifs). On note que *a..e* n'a pas été transformé car il y a une virgule entre accolades
4. `{{a..e}, fic}*` est remplacé par *a\*b\*c\*d\*e\* fic\** qui seront à l'expansion des chemins de fichiers (ce sont tous des motifs)

□

## 4.F Protection par chaînes

Le *backslash* n'est pas le seul caractère spécial permettant de protéger les autres caractères spéciaux. Il y a aussi les protections par chaînes en utilisant des ' (apostrophe ou *quote*) ou des " (guillemets ou *double-quote*).

### 4.F.1 Protection par quotes

L'apostrophe ' (ou *quote*) est un **constructeur de chaîne**. Elle s'utilise par paires : la première marque le début d'une chaîne et la suivante sa fin, dans laquelle **aucun caractère spécial n'est interprété** (même le *backslash*, les blancs ou les retours à la ligne). Elle est plus pratique (et plus lisible) à utiliser que le *backslash* lorsque plusieurs caractères spéciaux apparaissent dans la chaîne.

## Exemples

Alors que pour la commande :

```
$ ls -l *fic?.txt
```


*bash* remplacerait le motif par la liste de tous les fichiers dont le nom se termine par **fic** suivi d'un caractère quelconque puis **.txt**, et lancerait l'exécution de **ls** avec cette liste en arguments, pour la commande :

```
$ ls -l '*fic?.txt'
```

*bash* supprime simplement les apostrophes, n'interprète pas les caractères spéciaux et lance **ls -l \*fic?.txt** (on obtient alors les détails du fichier *\*fic?.txt*), ce qui revient au même que :


```
$ ls -l \*fic\?.txt
```

□

 Entre deux quotes, il ne peut pas y avoir une autre quote, même si elle est précédée d'un backslash (qui n'a plus de signification spéciale). Ainsi, écrire **ls 'c\'est un fichier'** provoque une erreur.

## 4.F.2 Protection par guillemets

Les guillemets " s'emploient comme l'apostrophe mais les caractères spéciaux suivants ne sont pas protégés entre guillemets : le dollar \$, l'apostrophe inversée ` et le point d'exclamation !. Le backslash perd aussi sa signification entre guillemets, sauf s'il précède l'un de ces caractères, un guillemet, un retour à la ligne ou un backslash, auquel cas il le protège.

 Les quotes et les guillemets peuvent être placés n'importe où dans la ligne de commande, y compris au milieu d'un mot.


 On aura compris que les apostrophes sont protégées entre guillemets et inversement.

### Exemples

*Les combinaisons de caractères de protection suivantes sont équivalentes et permettent d'obtenir les détails du fichier nommé `un_nom_bizarre*` :*

```
$ ls -l un\_nom\_bizarre\*
$ ls -l 'un_nom_bizarre*'
$ ls -l "un_nom_bizarre*"
$ ls -l un'\_'nom\_bizarre\*
$ ls -l 'un_n'o"m_bizarre*'
```

□

 Pour protéger une chaîne de caractères, on utilisera de préférence les apostrophes. Les guillemets ne doivent être utilisés que si l'on veut que *bash* traite les caractères \$, ` ou !.

# Chapitre 5

## Gestion des entrées-sorties

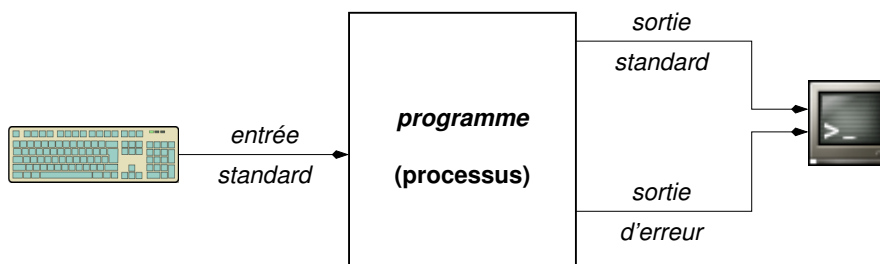
---

### 5.A Entrées-sorties par défaut

En principe, tout programme effectue un traitement sur des données et produit un résultat de ce traitement. Les données constituent en cela les entrées du programme. Le résultat en est une sortie :




Sous Unix, tout programme exécuté (appelé processus) dispose par défaut d'une entrée et de deux sorties : la sortie standard et la sortie d'erreur. Normalement, l'entrée correspond au clavier et les sorties correspondent à l'écran (ou fenêtre) :



et le clavier et l'écran sont tous deux désignés sous le nom générique de terminal : le clavier en entrée et l'écran en sortie.

La sortie standard est utilisée pour afficher les messages normaux, résultant du succès de la réalisation (de tout ou partie) du traitement demandé. C'est, par exemple, le chemin affiché lorsqu'on exécute la commande **pwd**, ou ce qui est affiché par la commande **ls**.

La sortie d'erreur est utilisée pour afficher les messages d'erreur lorsque le traitement n'a pas pu se dérouler normalement. C'est le cas du message affiché lorsqu'on tente d'accéder à un répertoire sur lequel on n'a pas le droit d'exécution : « *Permission non accordée* ». La sortie d'erreur est aussi parfois utilisée pour afficher certains messages d'interaction avec l'utilisateur, qui ne sont pas assimilables à un résultat du programme. Ces derniers sont, par exemple, les messages invitant l'utilisateur à taper quelque chose ou à faire un choix dans un menu, ou encore les messages de pause de la commande **less** (et de **more**) invitant à appuyer sur une touche. Ils n'apportent rien de plus au résultat affiché par les messages normaux.

 Puisque la sortie standard et la sortie d'erreur correspondent normalement à l'écran, les messages normaux et les messages d'erreur sont mélangés à l'écran.

## Exemples

Voici quelques exemples d'exécution de commandes utilisant les différentes sorties :

```
$ pwd
/home/toto
```

⇒ le chemin est affiché sur la sortie standard

```
$ echo Hello world
Hello world
```

⇒ le message est affiché sur la sortie standard. La commande **echo** sera vue dans la section [5.B.1](#).


```
$ ls -l fic
-rw-r--r-- 1 toto toto 101 2007-05-25 12:07 fic
```

⇒ les informations sur *fic* sont affichées sur la sortie standard

```
$ ls -l fic inconnu
ls: inconnu: Aucun fichier ou répertoire de ce type
-rw-r--r-- 1 toto toto 101 2007-05-25 12:07 fic
```

⇒ le message d'erreur concernant le nom *inconnu* est affiché sur la sortie d'erreur, et les informations sur *fic* sont affichées sur la sortie standard

□

 La plupart des commandes Unix (mais pas **ls** ni **echo** et toute la série de commandes **cp**, **mv**, etc.), prennent en entrée des fichiers passés en arguments ou, à défaut, leur entrée standard. Le résultat de leur traitement est le plus souvent affiché sur la sortie standard.

Il faut garder à l'esprit que même si un programme peut utiliser ce que l'utilisateur tape au clavier, il peut aussi utiliser des données provenant d'une autre source. Tout dépend du programme. Par exemple, il peut demander à l'utilisateur un nom de fichier contenant des données qu'il doit traiter. Il ouvrira ce fichier pour y puiser de l'information. Le fichier sera donc une entrée supplémentaire pour le programme.

De même, un programme peut utiliser l'écran pour afficher des messages, mais il peut aussi ouvrir ou créer des fichiers pour y placer de l'information.

## 5.B Affichage de texte et de fichiers

### 5.B.1 echo : afficher un message

La commande interne **echo** écrit à sur sa sortie standard, les arguments qui lui sont passés. Elle sera en particulier très utile lorsqu'on écrira des programmes bash mais aussi pour visualiser le contenu de variables, créer rapidement un fichier d'une ligne...

**i** Il existe aussi la commande **externe** **echo** correspondant au fichier exécutable `/bin/echo`. Elle agit de façon sensiblement différente et admet d'autres options. Pour avoir une aide sur la commande interne, il faut utiliser **help**. Pour une aide sur la commande externe, il faut utiliser **man**.

## Synopsis

**echo** [-neE] { *argument* }

Sans argument, écrit une ligne vide. Sinon, écrit une ligne contenant les *arguments* séparés par un espace. L'option **-n** demande de ne pas aller à la ligne.

## Exemples

```
$ echo hello world
hello world
```

⇒ les deux arguments (*hello* et *world*) sont écrits, séparés par un espace.

```
$ echo -n hello world
hello world$
```

⇒ **echo** n'écrit pas de retour à la ligne après le message et le prompt est affiché à la suite de celui-ci.

```
$ echo bonjour_les_amis
bonjour les amis
```

⇒ bien que les arguments soient séparés par plusieurs espaces, cela ne fait que trois arguments (*Bonjour, les et amis*) pour **echo**, qu'il écrit séparés par un seul espace.

```
$ echo "un fichier d'une ligne" > fic
```


⇒ crée *fic* contenant une ligne de texte grâce à une redirection de la sortie standard (voir section 5.C.2 page 68).

□

## Séquences d'échappement

L'option **-e** demande d'interpréter certaines séquences commençant par `\` (appelées séquences d'échappement) en les remplaçant par des caractères particuliers. Ces séquences et les caractères correspondants sont les suivants :

Séquence	Signification
<code>\a</code>	sonnerie (bell)
<code>\b</code>	recul d'un caractère (backspace)
<code>\e</code>	échappement (escape)
<code>\f</code>	saut de page (form feed)
<code>\n</code>	retour à la ligne (newline)
<code>\r</code>	retour charriot (carriage return)
<code>\t</code>	tabulation horizontale
<code>\v</code>	tabulation verticale
<code>\nnn</code>	caractère ASCII portant le code octal <i>nnn</i>

 Ces séquences commençant par `\`, elles doivent être protégées pour ne pas être interprétées par `bash`.

### Exemples

```
$ echo -e Une tabulation ici\t hop
Une tabulation ici      hop
```

⇒ *bash a lui-même interprété la séquence `\t` qui n'a pas de signification pour lui. Il l'a remplacée par `t` avant de lancer `echo` qui du coup n'a pas de séquence particulière en argument*


```
$ echo -e Une tabulation ici\\t hop
Une tabulation ici      \t hop
```

⇒ *bash a remplacé `\\` par `\` avant de lancer la commande. Du coup, `echo` traite `\t`.*

```
$ echo -e "Une tabulation ici\t hop"
Une tabulation ici      hop
```

⇒ *la séquence `\t` est protégée par les guillemets, donc `echo` écrit la tabulation*

□

 Si l'option de `bash` `xpg_echo` est active, il n'est pas nécessaire d'utiliser l'option `-e` pour que `echo` interprète les séquences d'échappement. Si elle est active mais qu'on ne veut pas que ces séquences soient interprétées, il faut utiliser l'option `-E` de `echo`.

## 5.B.2 cat : concaténer des fichiers


La commande externe `cat` va nous permettre d'illustrer certaines notions sur les entrées-sorties. Elle écrit sur sa sortie standard ce qu'elle lit en entrée. Elle est aussi beaucoup utilisée pour visualiser des petits fichiers.

### Synopsis

```
cat [-nv] {référence}
```

Elle ne fait qu'écrire sur sa sortie standard le contenu des fichiers ordinaires qui lui sont passés en arguments. Ils se trouvent ainsi **concaténés** (leur contenu est mis bout à bout, sans séparation) sur la sortie.

Sans argument, `cat` se contente d'écrire sur sa sortie standard ce qu'elle lit sur son entrée standard, jusqu'à rencontrer la marque de fin de fichiers (que l'on produit au clavier par la combinaison `CTRL-D`).

 Si `cat` a des arguments, alors elle ignore son entrée standard comme nous le verrons dans la section 5.C.1 page 67.

L'option `-n` demande d'afficher le numéro des lignes écrites. L'option `-v` demande d'afficher les caractères de contrôle (non imprimables) hors sauts de lignes ou tabulations en les faisant précéder de `^` ou de `M-`.

### Exemples

```
$ cat fic1
```

```
je suis la première ligne de fic1
je suis la dernière ligne de fic1
```

⇒ *affichage à l'écran du contenu du fichier `fic1` qui ne contient visiblement que deux lignes. Ces lignes sont écrites sur la sortie standard.*

```
$ cat fic1 inexistant
```

```
je suis la première ligne de fic1
je suis la dernière ligne de fic1
cat: inexistant: Aucun fichier ou répertoire de ce type
```

⇒ *la ligne en italiques est un message d'erreur de `cat`. Elle est écrite sur la sortie d'erreur (donc l'écran).*

```
$ cat sorties.cxx
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main (void)
```

```
    cout << "Ce texte a été écrit sur la sortie standard" << endl ;
    cerr << "Ce texte a été écrit sur la sortie d'erreur" << endl ;
    return 0;
```

```
// main()
```

⇒ *affichage à l'écran du contenu du fichier `sorties.cxx` qui contient visiblement un petit programme C++ qui utilise les flux `cout` et `cerr` pour écrire un message dans chaque sortie (standard et d'erreur).*

```
$ cat fic1 fic2 fic3
```

```
je suis la première ligne de fic1
je suis la dernière ligne de fic1
je suis la première ligne de fic2
je suis la dernière ligne de fic2
je suis la première ligne de fic3
je suis la dernière ligne de fic3
```

⇒ *Écriture des 3 fichiers à la suite, chacun constitué de deux lignes.*

❗ On aurait pu écrire `cat fic[1-3]`

```
$ cat
```

**Tout ce qui est tapé au clavier**

Tout ce qui est tapé au clavier

**est réécrit sur la sortie standard.**

est réécrit sur la sortie standard.

**on se demande bien à quoi**

on se demande bien à quoi

**cet exemple va servir...**

cet exemple va servir...

**CTRL-D**

⇒ *ici les caractères en gras sont lus sur l'entrée standard. Chaque ligne lue est copiée sur l'écran (les caractères non gras). La combinaison `CTRL-D` simule une fin de fichier (qui est attendue par `cat`).*

□

### 5.B.3 less : afficher le contenu de fichiers en paginant

Une commande très utile pour visualiser de grands fichiers est la commande externe **less**. En effet, celle-ci affiche le contenu des fichiers en paginant l’affichage.

#### Synopsis

**less** { référence }

Affiche, sur sa sortie standard et en paginant, le contenu des fichiers passés en arguments ou, à défaut, ce qu’elle lit sur son entrée standard jusqu’à lire `CTRL-D` (la marque de fin de fichier).

Pendant la pagination, **less** écrit en bas de la page un message comportant un pourcentage, si ce qu’il reste à lire est connu (ce n’est pas le cas si elle lit sur son entrée standard). Ce message d’état est en fait écrit sur sa sortie d’erreur. Elle attend alors que l’utilisateur tape l’une des commandes (de **less**) suivantes (consulter le manuel pour d’autres commandes) :

**q** pour quitter

**g** pour aller à la première ligne

**G** pour aller à la dernière ligne

`Entrée` pour faire défiler d’une ligne

`Espace` pour faire défiler d’une page

**b** pour revenir d’une page en arrière

**:n** `Entrée` pour faire afficher le fichier suivant

**:p** `Entrée` pour faire afficher le fichier précédent

**:f** `Entrée` pour faire afficher le nom du fichier courant et le numéro de ligne

**/expreg** `Entrée` pour rechercher vers l’avant une chaîne correspondant à l’expression régulière *expreg* (voir chapitre 12 page 151)

**?expreg** `Entrée` pour rechercher en arrière une chaîne correspondant à l’expression régulière *expreg*.

**i** La commande externe **more** est également disponible. Elle est plus ancienne que **less** et offre moins de fonctionnalités. Il est conseillé d’utiliser **less** si celle-ci est disponible.

#### Exemples


```
$ less /etc/passwd
```

⇒ affiche en paginant le contenu du fichier */etc/passwd* qui contient des informations sur tous les utilisateurs connus du système.

```
$ ls -l /usr/bin | less
```

⇒ affiche en paginant le contenu du répertoire */usr/bin* avec informations détaillées, grâce à l’utilisation d’un tube (voir chapitre 6)




 Certains se demanderont dans l'exemple précédent comment **less** peut-il lire les frappes au clavier qui le pilotent, alors que son entrée est redirigée ? Sûrement parce que le programmeur a prévu de lire ces frappes depuis le **terminal** (en lecture, c'est le clavier) et non pas son entrée.

(à méditer...)



## 5.C Redirections

Tous les interpréteurs de commandes sous Unix fournissent un mécanisme pour modifier les entrées et les sorties des commandes lancées. Cela s'avère très pratique car cela permet de faire en sorte qu'une commande lise ses entrées depuis un fichier et non le clavier, et/ou que ses sorties soient écrites dans des fichiers plutôt que l'écran. Cela permet aussi de fabriquer des commandes complexes, ou une commande utilisera comme entrée ce qu'une autre écrit sur une de ses sorties. Dans bash, cela se fait très facilement en utilisant les caractères spéciaux `<`, `>` et `|` (ainsi que quelques autres).

 Les redirections sont des moyens simples et élégants permettant de modifier (dérouter) l'entrée et/ou les sorties d'un programme et ceci de manière transparente pour celui-ci <sup>1</sup>.

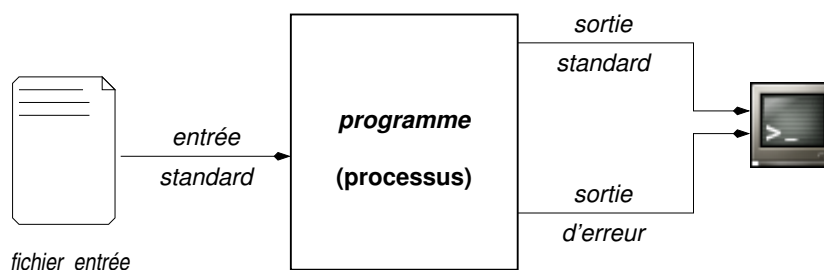
### 5.C.1 Redirection de l'entrée standard

L'entrée standard d'un programme est par défaut le clavier. Pour la dérouter afin qu'elle soit un fichier *fichier\_entrée*, il suffit de placer dans la ligne de commande, la séquence :

`< fichier_entrée`

où `<` est le caractère spécial ordonnant la redirection de l'entrée standard.

Ce que le programme devait lire à partir du clavier, il le lira désormais dans le fichier *fichier\_entrée*. On ne peut même plus se servir du clavier pour fournir des données au programme (mais on devrait toujours pouvoir l'arrêter en tapant `CTRL-C`) :



### Exemples

```
$ cat < fic1
je suis la première ligne de fic1
je suis la dernière ligne de fic1
```

1. Les programmes peuvent toutefois savoir s'il y a redirection et modifier leur comportement en conséquence. Par exemple, si la sortie de **ls** sans option est l'écran, alors les fichiers sont placés en colonnes. Si la sortie n'est pas l'écran, ils seront écrits un par ligne.

- ⇒ avant de lancer **cat**, **bash** fait en sorte que son entrée ne soit plus le clavier mais que ce soit *fic1*. Puisqu'il n'y a pas d'argument, **cat** adopte son comportement par défaut : il écrit sur sa sortie (l'écran), ce qu'il lit sur son entrée (*fic1*).

```
$ cat < fic1 fic2
```

```
je suis la première ligne de fic2
```

```
je suis la dernière ligne de fic2
```

- ⇒ ici, malgré la redirection de l'entrée, il reste *fic2* comme argument pour **cat**. Il ignore donc son entrée et écrit le contenu de *fic2*.

✍ Par soucis de lisibilité, il est préférable de placer les redirections à la fin de la commande. Ainsi, la commande précédente est plus lisible ainsi : **cat fic2 < fic1**.

□

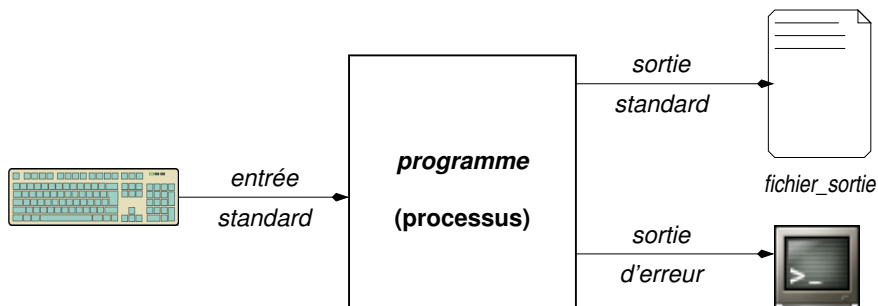
## 5.C.2 Redirection de la sortie standard

La sortie standard d'un programme est par défaut l'écran. Pour la dérouter afin qu'elle soit un fichier *fichier\_sortie*, il suffit de placer dans la ligne de commandes, la séquence :

> *fichier\_sortie*

où > est le caractère spécial ordonnant la redirection de la sortie standard.

Ceci a pour effet d'**écraser**<sup>2</sup> ou de **créer** le fichier *fichier\_sortie* afin qu'il contienne ce que le programme écrit sur sa sortie standard (et qui serait normalement affiché à l'écran) :



✍ Pour que cette redirection réussisse, il faut :

- si le fichier existe (écrasement), avoir la permission d'écriture sur ce fichier
- s'il n'existe pas (création), avoir la permission d'écriture sur le répertoire.

Dans tous les cas, il faut aussi avoir la permission d'exécution sur le répertoire. Notons que la redirection échoue si le fichier existe mais sans la permission d'écriture, même avec les permissions d'écriture et d'exécution sur le répertoire (qui permettraient de supprimer le fichier).

## Exemples

2. Si l'option *noclobber* est active, on ne peut écraser ainsi un fichier existant et cela produirait une erreur. On peut toutefois utiliser la redirection >| qui fonctionne comme > mais ignore *noclobber*.

```
$ cat fic1 > fic1.old
```

⇒ copie le contenu de *fic1* dans *fic1.old* (en le créant ou l'écrasant).

```
$ cat < fic1 > fic1.old
```

⇒ une autre façon de réaliser cette copie, en redirigeant aussi l'entrée standard.

```
$ cat fic* > nouveau
```

⇒ création du fichier *nouveau* qui contient la concaténation du contenu de tous les fichiers dont le nom commence par **fic**. Les fichiers sont concaténés dans l'ordre ASCII de leur nom (par exemple, *fic1* avant *fic1.old* puis *fic2...*).

```
$ cat fic1 inexistant > fic1.old
```

*cat: inexistant: Aucun fichier ou répertoire de ce type*

⇒ copie *fic1* dans *fic1.old* et le message d'erreur est laissé sur la sortie d'erreur.

```
$ cat > mon-texte.txt
```

Tout ce texte jusqu'à l'appui de la touche Ctrl-d sera écrit...

(même cette ligne)

...dans le fichier *mon-texte.txt*.

Magique, non ?

```
CTRL-D
```

```
$
```

⇒ sans argument, **cat** écrit sur sa sortie standard (ici, le fichier *mon-texte.txt*) ce qu'elle lit sur son entrée standard (ici le clavier).

□

Il est aussi possible de ne pas écraser le fichier s'il existe mais d'ajouter à la fin de celui-ci ce qui est écrit sur la sortie standard. Cela se fait ainsi :

```
>> fichier_sortie
```

Cette écriture a le même effet que la précédente si le fichier n'existe pas.

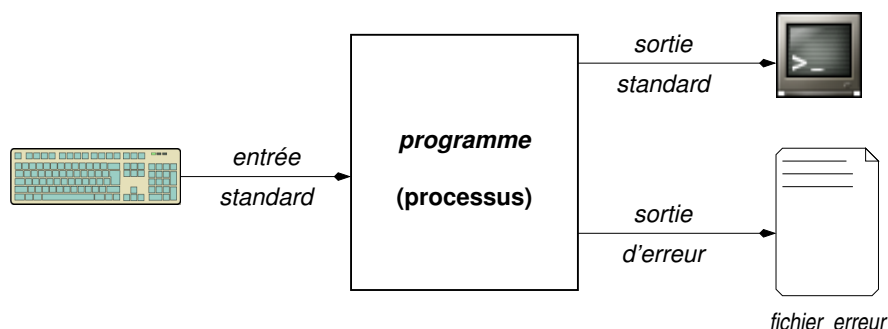
### 5.C.3 Redirection de la sortie d'erreur

La sortie standard d'erreur d'un programme est par défaut l'écran. Pour la dérouter afin qu'elle soit un fichier *fichier\_erreur*, il suffit de rajouter à la suite des arguments, la séquence :

```
2> fichier_erreur
```

où **2>** est une séquence spéciale ordonnant la redirection de la sortie d'erreur.

Ceci a pour effet d'**écraser**<sup>3</sup> ou de **créer** le fichier *fichier\_erreur* afin qu'il contienne ce que le programme écrit sur sa sortie d'erreur (et qui serait normalement affiché à l'écran).



3. L'option *noclobber* affecte aussi cette redirection. On ignore *noclobber* avec **2> |**.

 Cela laisse inchangé la sortie standard.

## Exemples

```
$ cat fic1 2> erreurs
```

```
je suis la première ligne de fic1
je suis la dernière ligne de fic1
```

⇒ *fic1 est écrit à l'écran et les messages d'erreur sont placés dans erreurs (en le créant ou l'écrasant). Ici, il n'y a pas d'erreur, donc erreurs sera vide.*

```
$ cat fic1 fic2 inexistant 2> erreurs
```

```
je suis la première ligne de fic1
je suis la dernière ligne de fic1
je suis la première ligne de fic2
je suis la dernière ligne de fic2
```

⇒ *Le message d'erreur indiquant l'inexistence de inexistant est placé dans erreurs. Le contenu de fic1 et fic2 est écrit à l'écran.*

□

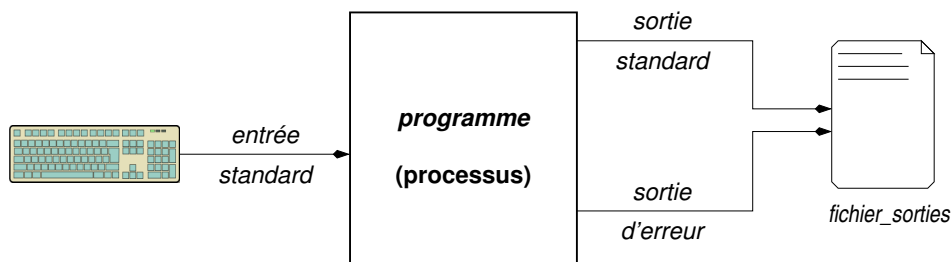
Il est aussi possible de ne pas écraser le fichier s'il existe mais d'ajouter à la fin de celui-ci ce qui est écrit sur la sortie standard. Cela se fait ainsi :

```
2>> fichier_erreur
```

Cette écriture a le même effet que la précédente si le fichier n'existe pas.

## 5.C.4 Redirection des sorties ensemble

Il y a plusieurs écritures permettant de rediriger la sortie standard et la sortie d'erreur dans un même fichier *fichier\_sorties* :



Les plus simples sont les suivantes :

```
&> fichier_sorties
ou
>& fichier_sorties
```

La première forme est à préférer. Ces écritures sont en fait un raccourci d'une forme plus complexe :

```
> fichier_sorties 2>&1
```

où l'on redirige d'abord la sortie standard puis l'on redirige la sortie d'erreur sur la même chose<sup>4</sup> que la sortie standard.

Pour ne pas écraser *fichier.sorties* il faudra utiliser la séquence suivante :

`&>> fichier.sorties`

qui est équivalente à la forme plus complexe :

`>> fichier.sorties 2>&1`

## 5.D tty : connaître son terminal

Le clavier et l'écran sont regroupés sous le terme générique de terminal. Un terminal est mis à disposition de l'utilisateur à chaque fois que celui-ci ouvre une fenêtre texte nécessitant des entrées au clavier et des sorties à l'écran (de type **aterm** ou **gnome-terminal**, celle où **bash** est exécuté). Pour connaître le terminal en cours d'utilisation, il suffit de taper la commande externe **tty** sans option ni argument.

**i** Le terminal d'un utilisateur est un fichier spécial, dans lequel il est possible d'écrire (selon ses permissions). Tout ce qui y est écrit est affiché sur la fenêtre utilisant ce (fichier) terminal.

### Exemples

```
cpb$ tty
/dev/pts/1
cpb$ ls -l /dev/pts/1
crw----- 1 cpb      tty      136,   1 sep 19 05:42 /dev/pts/1
```

⇒ on remarque le **c** en début de ligne, qui caractérise un fichier (spécial) de périphérique en mode caractère, que cpb (l'utilisateur en session) en est le propriétaire et possède les droits de lecture et d'écriture.

```
cpb$ echo Salut > /dev/pts/1
```

⇒ exécuté dans une fenêtre quelconque, cela a pour effet d'écrire *Salut* dans la fenêtre où a été tapée **tty**.

□

**i** Il existe un terminal fictif `/dev/null`, sur lequel tout le monde a les droits de lecture et d'écriture, mais qui ne représente rien : tout ce qui est écrit dedans est perdu et rien ne peut y être lu. Ce fichier est principalement utilisé comme destination de la redirection d'une sortie d'une commande afin de se débarrasser des informations (inutiles) qu'elle y écrit.

### Exemple

```
$ echo Salut > /dev/null
```

⇒ n'importe qui peut exécuter cette commande qui n'a aucun effet car *Salut* n'est écrit nulle part.

□

4. `2>&1` est une instance de la redirection `n>&m`, indiquant que la sortie *n* (2 représente la sortie d'erreur) devient la même que la sortie *m* (1 représente la sortie standard).



# Chapitre 6

## Tubes et filtres

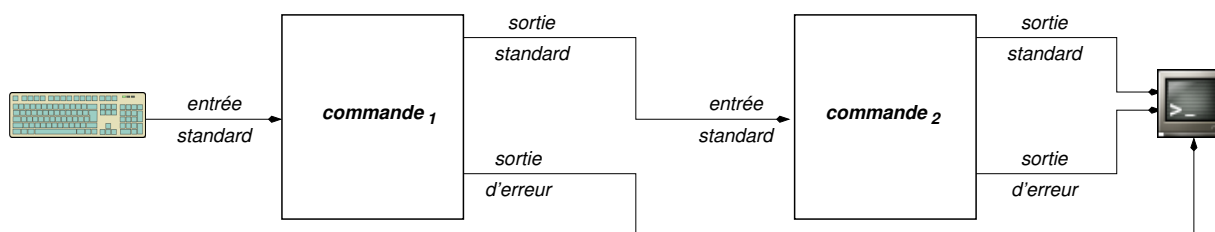
### 6.A Tubes (pipes)

Les **tubes** (*pipes*) font partie du fondement d'Unix. Ils permettent de former des chaînes de traitement complexes en cascasant des commandes, de manière à ce qu'une sortie d'une commande devienne l'entrée d'une autre. Ainsi, tout ce qu'une commande écrit pourra être lu sur l'entrée standard de l'autre commande. Le tube réalise donc aussi une redirection des entrées-sorties.

On crée un tube en utilisant le caractère spécial `|` (*pipe*). Ainsi :

`commande1 | commande2`

déclenche l'exécution simultanée des deux commandes et la sortie standard de `commande1` devient l'entrée standard `commande2` :



Il est possible d'avoir plusieurs pipes sur la même ligne de commande, pour former ce qu'on appelle un **pipeline** :

`commande1 | commande2 | ... | commanden`

Le principe d'un pipeline est le suivant : la `commande1` doit écrire un résultat sur sa sortie standard. Au fur et à mesure qu'il est écrit, il est traité en entrée par la `commande2` qui elle-même produit un résultat sur sa sortie standard, qui est traité par la commande suivante du pipeline, etc. En ce sens, les pipes permettent d'écrire des commandes complexes et inédites.

#### Exemples

```
$ ls -l /etc | less
```

⇨ affiche en paginant des informations détaillées sur les fichiers du répertoire `/etc`.

```
$ cat fic1 fic2 fic3 | wc -l
```

6

⇒ compte le nombre de lignes total des fichiers *fic1*, *fic2* et *fic3* concaténés. Voir section 6.B.1 page suivante pour une description de la commande **wc**.

```
$ cat < fic1 | cat | cat > fic1.old
```

⇒ Une façon (bien particulière, cependant) de copier le fichier *fic1* en *fic1.old*.

□



Une commande ne doit être utilisée dans un pipeline que si elle lit sur son entrée standard les données qu'elle doit traiter, à moins que ce ne soit la première commande du pipeline. Par exemple, une commande comme **ls** ou **echo** ne devrait figurer que comme première commande du pipeline. Les commandes muettes ne devraient figurer que comme dernière commande d'un pipeline. Les commandes ne lisant rien sur leur entrée standard et en plus muettes (comme **cd**, **chmod**, **rm**, ...) ne devraient même pas figurer dans un pipeline.

## Placer les deux sorties dans un tube

Pour rediriger aussi la sortie d'erreur par le pipe entre deux commandes, il faut écrire :

```
commande1 2>&1 | commande2
```



La logique voudrait que l'on écrive plutôt **| 2>&1** mais le **|** est un séparateur de commandes. C'est pourquoi la redirection de la sortie d'erreur doit être placée **avant** le tube.

## 6.B Filtres

On appelle **filtre**, une commande capable de lire des données (souvent du texte) sur son entrée standard et qui écrit le résultat de son traitement sur la sortie standard. Cette définition assez vague caractérise les commandes pouvant être placées dans un pipeline.

Or selon la philosophie Unix, les utilitaires doivent être spécialisés dans une tâche, mais doivent faciliter leur insertion dans une chaîne de traitement, et en particulier un pipeline. Ainsi, de nombreux utilitaires Unix peuvent prendre place dans un pipeline. La composition de leurs fonctions, aussi limitées soient-elles, conduit à des traitements complexes avec une simple ligne de commande. Parmi ces utilitaires, on peut citer :

**cat** reproduire l'entrée sur la sortie ;

**tac** reproduire l'entrée sur la sortie, en inversant l'ordre des lignes ;

**shuf** reproduire l'entrée sur la sortie, avec un ordre des lignes aléatoire ;

**tee** reproduire l'entrée sur la sortie et dans des fichiers ;

**less** paginer l'entrée (en principe en fin de pipeline) ;

**nl** numéroté les lignes de l'entrée sur la sortie ;

**paste** coller les lignes de même numéro des fichiers d'entrée sur la sortie ;

**head** ne garder que les premières lignes en entrée ;

**tail** ne garder que les dernières lignes en entrée ;

**uniq** éliminer des lignes identiques ;



**sort** trier les lignes de l'entrée ;  
**grep** filtrer l'entrée en ne gardant que les lignes contenant certaines chaînes ;  
**sed** modifier les lignes lues en entrée ;  
**tr** remplacer des caractères de l'entrée par d'autres caractères ;  
**cut** extraire des colonnes sur les lignes en entrée ;  
**awk** programmer un traitement des lignes d'entrée ;  
 ...

Certains d'entre eux font l'objet d'une description dans ce document, notamment dans ce qui suit.

### Exemples

La commande **less** est souvent employée à la fin d'un pipeline lorsque le résultat de la commande qui la précède dans le pipeline est trop long pour tenir sur une page. Il en découle une pagination du résultat de la commande précédente :

```
$ ls -l /bin | less
```

⇒ *pagine la sortie de **ls** qui afficherait sinon d'un coup plusieurs dizaines de lignes. Dans le cas d'un terminal de 24 lignes sans fenêtre ni ascenseur, il ne serait pas possible de voir le début de sa sortie.*

La pagination de l'affichage est l'un des cas où il peut-être utile de rediriger aussi la sortie d'erreur dans le pipeline. Par exemple, la commande **ls -lR /** exécutée par un utilisateur normal devrait afficher de nombreux messages d'erreur qu'il peut être intéressant de paginer avec le reste de la sortie :

```
$ ls -lR / 2>&1 | less
```

□

## 6.B.1 wc : compter les lignes, mots et octets de fichiers

La commande externe **wc** permet d'effectuer des comptages sur un ou plusieurs fichiers.

### Synopsis

```
wc [-cwlL] { référence }
```

Les options précisent les éléments à compter :

- l'option **-l** demande le comptage des lignes ;
- l'option **-w** celui des mots ;
- l'option **-c** celui des octets ;
- l'option **-L** demande d'indiquer la taille de la plus longue ligne ;
- l'option **-m** a été récemment introduite : elle demande de compter les caractères. Elle n'est utile que si le fichier est codé avec un jeu de caractère étendu (par exemple, en utf-8 un caractère peut occuper jusqu'à 4 octets).

Si aucune option n'est précisée, **wc** active par défaut les options **-l**, **-w** et **-c**.

Pour chaque fichier *référence* indiqué, **wc** écrit une ligne contenant les comptages précisés demandés par les options suivis du nom du fichier. Elle a toujours la forme suivante :

*lignes mots octets caractères maxlongueur référence*

La présence des différents comptages dépend des options.

Si plusieurs *références* sont indiquées une ligne de totalisation est écrite à la fin et a la forme suivante :

*total-lignes total-mots total-octets total-caractères maxlongueur total*

Si aucune *référence* n'est indiquée, **wc** effectue le(s) comptage(s) sur ce qu'elle lit depuis l'entrée standard.

## Exemples

```
$ wc /etc/passwd
```

```
282 807 20015 /etc/passwd
```

⇒ Il y a 282 lignes (donc a priori 282 utilisateurs recensés), 807 mots et 20015 octets dans le fichier /etc/passwd.

```
$ wc -l /etc/passwd
```

```
282 /etc/passwd
```

⇒ N'affiche que le nombre de lignes du fichier.

```
$ wc -lL /etc/passwd /etc/group
```

```
282 81 /etc/passwd
```

```
289 999 /etc/group
```

```
571 999 total
```

⇒ On obtient le comptage des lignes et la taille de la plus longue ligne pour chaque fichier, et une ligne de totalisation.

🔗 On remarque que le nom des fichiers soumis au comptage est affiché. Pour éviter cela, il faut empêcher que **wc** ait les fichiers en arguments comme avec les commandes suivantes.

```
$ cat /etc/passwd | wc -l
```

```
282
```

⇒ Seul 282 est affiché.

```
$ cat /etc/passwd /etc/group | wc -lL
```

```
571 999
```

⇒ On obtient ici les mêmes informations que la ligne de totalisation (sans *total* à la fin).

```
$ ls /home | wc -l
```

```
230
```

⇒ Il y a 230 fichiers (ou répertoires) dans le répertoire /home

```
$ cat fic_utf8
```

```
àç€
```

```
$ file fic_utf8
```

```
fic_utf8: UTF-8 Unicode text
```

⇒ Le fichier *fic\_utf8* est codé en **utf8**. Il contient 4 caractères : à, ç, € et le retour à la ligne.

```
$ wc -c fic_utf8
```

```
8 fic_utf8
```

⇒ *il contient en tout 8 octets : à et ç sont codés sur 2 octets, € est codé sur 3 octets et le retour à la ligne sur un seul.*

```
$ wc -m fic_utf8
4 fic_utf8
```

⇒ *Avec l'option -m, on constate que le fichier contient bien 4 caractères.*

□

## 6.B.2 head : écrire le début de fichiers

La commande externe **head** écrit sur sa sortie standard le début des fichiers indiqués en arguments ou, à défaut, de l'entrée standard.

### Synopsis

```
head [-n nombre | -c nombre] [-q | -v] {référence}
```

Les options **-n** et **-c** sont exclusives, de même que les options **-q** et **-v**. Néanmoins, leur présence simultanée ne produit pas d'erreur : la plus à droite sur la ligne de commande est celle retenue.

Sans option, **head** affiche les 10 premières lignes de chaque fichier *référence* en arguments. L'option **-n** permet d'indiquer le nombre de lignes souhaitées, alors que l'option **-c** indique le nombre d'octets souhaités.

*nombre* est de la forme  $[-] n[\mathbf{b}|\mathbf{k}|\mathbf{m}]$  où *n* est une valeur numérique indiquant le nombre de lignes ou d'octets. Si *n* est précédé de **-**, **head** écrira tout sauf les *n* dernières lignes/derniers octets des fichiers. *n* peut être suivi d'un suffixe (**b**, **k** ou **m**) qui est un multiplicateur (**b** pour  $\times 512$ , **k** pour  $\times 1024$  et **m** pour  $\times 1024^2$ ).

Si plusieurs *références* sont indiquées, **head** affichera un en-tête avant chaque contenu de fichier écrit. Cet en-tête a la forme **==> référence <==**. Entre deux fichiers, l'en-tête est précédé d'un passage à la ligne. L'affichage de l'en-tête est désactivable par l'option **-q**. Il est forcé (dans le cas d'un seul fichier d'entrée) par l'option **-v**.

Si aucune *référence* n'est indiquée **head** ne traite que l'entrée standard. Si une *référence* est **-**, **head** traite l'entrée standard en plus de ceux indiqués.

### Exemples

```
$ ls -l
total 8
-rw-r--r-- 1 cyril users 24 2008-02-07 11:41 fic1
-rw-r--r-- 1 cyril users 24 2008-02-07 11:41 fic2
$ ls -l | head -n 1
total 8
```

⇒ *seule la première ligne de la sortie de ls est affichée*

```
$ ls -l | head -n -1
total 8
-rw-r--r-- 1 cyril users 24 2008-02-07 11:41 fic1
```

⇒ *la dernière ligne de la sortie de ls n'est pas écrite*

```
$ head fic1 fic2
```

```
==> fic1 <==
```

```
aaa
bbb
ccc
ddd
eee
fff
```

```
==> fic2 <==
```

```
111
222
333
444
555
666
```

⇒ les 10 premières lignes des fichiers *fic1* et *fic2* sont affichées, un en-tête pour chaque fichier.

```
$ head -n 2 -q fic1 fic2
```

```
aaa
bbb
111
222
```

⇒ les 2 premières lignes de chaque fichier sont écrites, sans en-tête

```
$ head -c 2 -q fic1 fic2
```

```
aa11$
```

⇒ les 2 premiers octets de chaque fichier sont écrits, sans en-tête. Le prompt est affiché sans passage à la ligne.

□

### 6.B.3 tail : écrire la fin de fichiers

La commande externe **tail** fait l'inverse de **head** : elle écrit sur sa sortie standard la fin des fichiers indiqués en arguments ou, à défaut, de l'entrée standard. Elle admet cependant un peu plus d'options surtout utiles pour le suivi des journaux système.

#### Synopsis

```
tail [-n nombre | -c nombre] [-q | -v] [options-de-suivi] {référence}
```

**tail** s'utilise de la même manière que **head** : les options **-n** et **-c** sont exclusives, de même que les options **-q** et **-v**. Elles ont la même signification que pour **head**, mais les options **-n** et **-c** spécifient le nombre des dernières lignes/derniers octets à afficher. Comme **head**, les options **-q** et **-v** permettent de désactiver et de forcer l'affichage des en-têtes. Le traitement des *références* est le même que pour **head**, notamment si une *référence* est **-** l'entrée standard sera traitée.

*nombre* est de la forme **[+]** *n* [**b** | **k** | **m**]. Elle diffère de celle pour **head** uniquement par le premier caractère optionnel : le signe **+** placé avant *n* indique que l'on veut afficher de la ligne/octet *n* jusqu'à la fin des fichiers. Sans option, **tail** affiche les 10 dernières lignes de chaque fichier *référence* en arguments.

**i** **tail** est souvent utilisée par les administrateurs système pour consulter en temps réel les fichiers journaux du système (appelés les *logs*). Ces fichiers se trouvent généralement dans le répertoire `/var/log`. Ce sont des fichiers texte (d'extension `.log`) qui sont alimentés par le noyau (*kernel*) et les démons<sup>1</sup> qui leur ajoutent des lignes de compte-rendu.

Les *options-de-suivi* sont particulièrement adaptées à la consultation des journaux système. Plutôt que de se terminer après l'écriture des dernières lignes/derniers octets des fichiers, **tail** va se placer en attente et consulter régulièrement le(s) fichier(s) et afficher les lignes nouvelles (même si on choisit l'option **-c**) qui leur sont ajoutées.

Des *options-de-suivi* possibles sont :

- f** pour activer le suivi
- pid=pid** demande de se terminer lorsque le processus de PID *pid* se termine

D'autres *options-de-suivi* existent, en particulier pour tenir compte du fait que les journaux système sont régulièrement renommés pour diverses raisons : redémarrage du démon qui alimente un log particulier, taille limite atteinte,... Par exemple, `syslog` va devenir `syslog.0` qui devient `syslog.1`, etc. et un nouveau fichier `syslog` sera créé.

## Exemples

```
$ ls -l
total 8
-rw-r--r-- 1 cyril users 24 2008-02-07 11:41 fic1
-rw-r--r-- 1 cyril users 24 2008-02-07 11:41 fic2
-rw-r--r-- 1 cyril users 26 2008-02-07 11:42 fic3
-rw-r--r-- 1 cyril users 26 2008-02-07 11:42 fic4
drwxr-xr-x 1 cyril users 4096 2008-02-07 11:43 rep1
drwxr-xr-x 1 cyril users 4096 2008-02-07 11:43 rep2
$ ls -l | tail -n+2
-rw-r--r-- 1 cyril users 24 2008-02-07 11:41 fic1
-rw-r--r-- 1 cyril users 24 2008-02-07 11:41 fic2
-rw-r--r-- 1 cyril users 26 2008-02-07 11:42 fic3
-rw-r--r-- 1 cyril users 26 2008-02-07 11:42 fic4
drwxr-xr-x 1 cyril users 4096 2008-02-07 11:43 rep1
drwxr-xr-x 1 cyril users 4096 2008-02-07 11:43 rep2
```

⇨ affiche tout sauf la première ligne (donc suppression de la ligne *total*)

```
$ ls -l | head -n 3 | tail -n 1
-rw-r--r-- 1 cyril users 24 2008-02-07 11:41 fic2
```

⇨ En combinant **head** et **tail**, on ne garde que la 3<sup>e</sup> ligne de la sortie de **ls**.

```
# tail -n 3 -f /var/log/syslog
Feb  8 11:49:48 incal hddtemp[3569]: /dev/hda: FUJITSU MHV2080AH: 48 C
Feb  8 11:50:48 incal hddtemp[3569]: /dev/hda: FUJITSU MHV2080AH: 48 C
Feb  8 11:51:48 incal hddtemp[3569]: /dev/hda: FUJITSU MHV2080AH: 48 C
Feb  8 11:52:48 incal hddtemp[3569]: /dev/hda: FUJITSU MHV2080AH: 48 C
```

CTRL-C

1. Un démon (*daemon*) est un processus exécuté en tâche de fond et qui a une certaine fonction : ce peut-être un serveur web (comme apache), un processus qui détecte le branchement de médias amovibles (clés USB, CD/DVD,...), etc.

⇒ tapée par root (car les utilisateurs ne peuvent pas consulter *syslog*), affiche les 3 dernières lignes de */var/log/syslog* et affiche les lignes qui lui seront ajoutées. La quatrième ligne (en italiques) a été ajoutée par la suite par le démon **hddtemp** qui récupère la température des disques durs. On utilise ensuite `CTRL-C` pour terminer **tail**.

□

## 6.B.4 cut : écrire des parties de lignes

La commande externe **cut** écrit sur sa sortie standard des parties de chaque ligne lue depuis les fichiers indiqués en arguments, ou à défaut sur l'entrée standard.

### Synopsis

```
cut -c intervalles [--output-delimiter=odélim] [--complement] {référence}
```

```
cut -f intervalles [-d délim] [-s] [--output-delimiter=odélim] [--complement] {référence}
```

Les options **-c** et **-f** sont mutuellement exclusives, et les options **-d** et **-s** ne sont disponibles que pour l'option **-f**.

L'option **-c** demande de n'afficher que les caractères précisés par les *intervalles*, où *intervalles* est une liste d'intervalles séparés par des virgules et a la forme générale suivante :

$$n-m\{, n-m\}$$

Un intervalle peut être incomplètement spécifié : *n-* veut dire du caractère *n* jusqu'à la fin de la ligne, et *-m* veut dire du début de la ligne jusqu'au caractère *m* (inclus). Si les intervalles se chevauchent, les caractères ne sont écrits qu'une fois. Les caractères sont écrits dans leur ordre d'apparition sur la ligne, quel que soit l'ordre des intervalles. Le premier caractère d'une ligne porte le numéro 1.

L'option **--output-delimiter** demande d'afficher le caractère précisé par *odélim* entre les intervalles qui ne se chevauchent pas.

L'option **-f** demande de traiter chaque ligne comme une série de champs (*fields*), où le séparateur de champ est indiqué par le caractère *délim* de l'option **-d**. Sans l'option **-d**, le délimiteur de champ par défaut est la tabulation. Si une ligne est dépourvue de délimiteur de champ, elle est écrite en entier, à moins que l'option **-s** soit précisée auquel cas elle est ignorée.

Les intervalles de **-f** ont la même forme que pour l'option **-c**. Le premier champ porte le numéro 1. Si une ligne en entrée comporte moins de champs que ceux demandés (par exemple contient 2 champs alors qu'on demande l'affichage du champ 3), alors une ligne vide est écrite en sortie. Les champs ne sont écrits qu'une fois et dans leur ordre d'apparition sur la ligne d'entrée.

L'option **--output-delimiter** demande de séparer les champs écrits par le caractère *odélim*. Si elle n'est pas précisée, le délimiteur en sortie est le même que celui en entrée.

L'option **--complement** inverse le sens de **-c** ou **-f** : elle demande l'affichage de tous les caractères ou champs qui ne sont pas ceux spécifiés.



L'option **--complement** n'est pas toujours disponible, mais l'est sur Debian.

Les *références* sont les fichiers à traiter en entrée. Si aucune *référence* n'est spécifiée, **cut** traite ce qu'il lit sur l'entrée standard. Une *référence* peut être *-* auquel cas **cut** traite aussi l'entrée standard en plus des autres fichiers indiqués.

❗ À noter que **cut** admet l'option **-b** qui s'utilise comme **-c** mais précise les octets et non les caractères. Pour le moment, elle donne le même résultat que **-c** mais cela changera lorsque l'internationalisation sera pleinement supportée, car selon le codage un caractère peut occuper plusieurs octets (jusqu'à 4 en utf-8).

### Exemple (Utilisation de l'option -c)

Soit le résultat de la commande suivante :

```
$ ls -l
total 1300
drwxr-xr-x  2 cpb prof  4096 jui  7  2005 boxes
drwxr-xr-x  3 cpb prof  4096 mar 24  2005 chat
-rwxr-xr-x  1 cpb prof 97404 mai  7  2002 checksum
-rwxr-xr-x  1 cpb prof   31 jun  4  2003 dataglu
drwxr-xr-x  2 cpb prof  4096 jui  7  2005 figlet
-rw-r--r--  1 cpb prof 834994 jun  8  2005 Firefox_wallpaper.png
drwxr-xr-x  9 cpb prof  4096 sep 13 09:07 unix
-rwxr-xr-x  1 cpb prof   76 jun  4  2003 viewdtg
```

On voudrait extraire uniquement les permissions et les noms de fichiers. On s'occupera plus tard d'éliminer la première ligne *total*. En remarquant que les permissions sont contenues dans les caractères 2 à 10 et que les noms de fichiers commencent au caractère 44, on peut utiliser **cut** pour extraire ce que l'on veut :



**La position des noms de fichiers peut varier car certaines informations affichées par ls occupent une place variable (nombre de liens, propriétaire, groupe, taille du fichier, ...)**

```
$ ls -l | cut -c2-10,44-
otal 1300
rwxr-xr-xboxes
rwxr-xr-xchat
rwxr-xr-xchecksum
rwxr-xr-xdataglu
rwxr-xr-xfiglet
rw-r--r--Firefox_wallpaper.png
rwxr-xr-xunix
rwxr-xr-xviewdtg
```

➡ on obtient bien ce que l'on veut mais les permissions et les noms de fichiers ne sont pas séparés. Pour les séparer d'un espace on va utiliser l'option **--output-delimiter** comme suit :

```
$ ls -l | cut -c2-10,44- --output-delimiter=' '
otal 1300
rwxr-xr-x boxes
rwxr-xr-x chat
rwxr-xr-x checksum
rwxr-xr-x dataglu
rwxr-xr-x figlet
rw-r--r-- Firefox_wallpaper.png
rwxr-xr-x unix
rwxr-xr-x viewdtg
```

Enfin, on élimine la première ligne à l'aide de la commande **tail** (ce qu'on aurait pu faire avant d'utiliser **cut**) :

```
$ ls -l | cut -c2-10,44- --output-delimiter='_' | tail -n+2
rwxr-xr-x boxes
rwxr-xr-x chat
rwxr-xr-x checksum
rwxr-xr-x dataglu
rwxr-xr-x figlet
rw-r--r-- Firefox_wallpaper.png
rwxr-xr-x unix
rwxr-xr-x viewdtg
```



### Exemple (Utilisation de l'option -f)

Soit le résultat (partiel) de la commande suivante :

```
$ cat /etc/group
root:x:0:
daemon:x:1:
dialout:x:20:toto,tata,titi,tutu
cdrom:x:24:toto,tata,titi,tutu
audio:x:29:tata,titi
video:x:44:tata,titi
users:x:100:
grp1:x:1001:toto,titi
grp2:x:1002:tata,tutu
grp3:x:1003:toto,tata,titi
```

Ce fichier contient les groupes et diverses informations séparées par des `:`. On voudrait extraire les groupes et leurs membres additionnels. Ces informations constituent le champ 1 et le champ 4. On utilise **cut** pour cela :

```
$ cut -d: -f1,4 /etc/group
root:
daemon:
dialout:toto,tata,titi,tutu
cdrom:toto,tata,titi,tutu
audio:tata,titi
video:tata,titi
users:
grp1:toto,titi
grp2:tata,tutu
grp3:toto,tata,titi
```

On souhaite maintenant séparer les membres additionnels par des espaces. Ceux-ci peuvent être vus comme des champs séparés par des virgules. Il suffit alors de traiter le résultat précédent par une nouvelle commande **cut** en demandant d'afficher tous les champs séparés par des virgules et de les écrire séparés par des espaces :

```
$ cut -d: -f1,4 /etc/group | cut -d, -f 1- --output-delimiter=' '
root:
daemon:
dialout:toto tata titi tutu
cdrom:toto tata titi tutu
audio:tata titi
video:tata titi
```



```
users:
grp1:toto titi
grp2:tata tutu
grp3:toto tata titi
```

Enfin, en remarquant que **dans ce cas précis** les groupes possédant des membres additionnels en possèdent plus de 2 et que ceux-ci étaient séparés par des virgules, on peut utiliser l'option **-s** afin d'éliminer les groupes qui n'en possèdent pas :

```
$ cut -d: -f1,4 /etc/group | cut -d, -f 1- -s --output-delimiter='_'
dialout:toto tata titi tutu
cdrom:toto tata titi tutu
audio:tata titi
video:tata titi
grp1:toto titi
grp2:tata tutu
grp3:toto tata titi
```

⇒ Si un groupe ne possédait qu'un membre additionnel, il aurait été aussi éliminé. Dans ce cas, il aurait fallu passer par un filtre de type **sed** ou **grep**, vus plus loin.

□

### 6.B.5 sort : trier des lignes de fichiers

La commande externe **sort** écrit sur sa sortie standard les lignes triées des fichiers qui lui sont passés en arguments, ou à défaut de ce qui est lu sur l'entrée standard.

#### Synopsis

```
sort [-nrhM] [-t séparateur] {-k pos1 [, pos2] } {fichier}
```

Les arguments *fichier* doivent être des fichiers texte. Le résultat du tri de l'ensemble des fichiers est écrit sur la sortie standard. Si aucun *fichier* n'est mentionné ou s'il vaut **-**, l'entrée standard est lue. Par défaut, le tri est effectué dans l'ordre croissant et les lignes sont comparées selon le codage des caractères qui les composent. Les options permettent d'exprimer des critères différents.

#### Exemples

```
$ cat noms1
Golade Larry 13600
Patagueul James 13001
Aidubois Laure 13013
Epipiolli Jeff 83140
$ sort noms1
Aidubois Laure 13013
Epipiolli Jeff 83140
Golade Larry 13600
Patagueul James 13001
$ cat noms2
MAGNE CHARLES 84150
BON JEAN 13001
AMETTO LUCIE 13008
PEUPLU JEAN 13600
$ sort noms1 noms2
```

```
Aidubois Laure 13013
AMETTO LUCIE 13008
BON JEAN 13001
Epipiolli Jeff 83140
Golade Larry 13600
MAGNE CHARLES 84150
Patagueul James 13001
PEUPLU JEAN 13600
```

□

Le tri précédent suit l'ordre alphabétique et est tout à fait correct. Cependant, il faut noter que cet ordre alphabétique n'est pas l'ordre classique ASCII qui repose uniquement sur la valeur du code ASCII des caractères et pour lequel chiffre < majuscules < minuscules. L'ordre alphabétique mélange les majuscules, les minuscules et les caractères accentués, et dépend des paramètres régionaux.

 Pour trier selon l'ordre classique, il faut faire précéder la commande de **LC\_ALL=C** (voir ci-dessous).

### Exemple


```
$ LC_ALL=C sort noms1 noms2
AMETTO LUCIE 13008
Aidubois Laure 13013
BON JEAN 13001
Epipiolli Jeff 83140
Golade Larry 13600
MAGNE CHARLES 84150
PEUPLU JEAN 13600
Patagueul James 13001
```

➡ Dans l'ordre classique, **AM** vient avant **Ai**, de même que **PE** vient avant **Pa**...

□

Plusieurs options permettent de modifier le comportement de **sort** ainsi que la clé de tri (qui est par défaut la ligne entière) :

- l'option **-r** inverse l'ordre de tri qui devient décroissant ;
- l'option **-n** demande d'utiliser le nombre figurant en début de ligne comme clé de tri principale. Ce nombre peut être décimal et signé ;
- l'option **-h** demande de traiter le premier mot comme une quantité humainement lisible (telle que 500K, 1M, 3, 2G...) et de l'utiliser comme clé de tri principale ;
- l'option **-M** demande de traiter le premier mot comme l'abréviation d'un mois et de l'utiliser comme clé de tri principale ;
- l'option **-f** demande d'ignorer la casse (ne pas faire de distinction entre majuscules et minuscules).

 Le séparateur de milliers, le caractère marquant la décimale et les abréviations de mois dépendent de la locale utilisée. Pour la locale française (fr\_FR.utf8) les mois abrégés et leur ordre sont :

janv. < févr. < mars < avril < mai < juin < juil. < août < sept. < oct. < nov. < déc.

### Exemples

```
$ cat fic
05:BB:janv.
1:AAA:juin
0:c:JANV.
10:c:févr.
1:dd:mai
8:AAA:déc.
-3:eee:mai
```

```
$ sort fic
05:BB:janv.
0:c:JANV.
10:c:févr.
1:AAA:juin
1:dd:mai
-3:eee:mai
8:AAA:déc.
```

⇨ *tri par ordre alphabétique croissant des lignes entières (dans la locale française, le tiret est ignoré)*

```
$ sort -n fic
-3:eee:mai
0:c:JANV.
1:AAA:juin
1:dd:mai
05:BB:janv.
8:AAA:déc.
10:c:févr.
```

⇨ *tri par ordre numérique croissant du premier nombre de la ligne*

```
$ sort -rn fic
10:c:févr.
8:AAA:déc.
05:BB:janv.
1:dd:mai
1:AAA:juin
0:c:JANV.
-3:eee:mai
```

⇨ *tri par ordre numérique **décroissant** du premier nombre de la ligne*

Utilisation de l'option -h

```
$ du -sh *
12K      fic1
4,0K     fic2
9,8M     rep1
687M     rep2
12K      rep3
4,4M     rep4
824K     fic3

$ du -sh * | sort -h
4,0K     fic2
12K      fic1
12K      rep3
824K     fic3
```

```
4, 4M      rep4
9, 8M      rep1
687M      rep2
```

⇒ tri croissant selon les quantités en début de ligne.



## Tri sur des champs et tri multi-critères

Jusqu'à présent, seul le premier mot/nombre ou la ligne entière était utilisé comme clé de tri. Les options **-t** et **-k** permettent d'utiliser une autre partie de la ligne comme clé de tri, et d'exprimer des critères multiples. Avec l'option **-k**, une ligne est considérée comme un ensemble de champs, et des champs sont sélectionnés comme clés de tri. Par défaut le séparateur de champ est un blanc. L'option **-t séparateur** fixe *séparateur* comme séparateur de champs à la place d'un blanc.

L'option **-k pos<sub>1</sub> [ , pos<sub>2</sub> ]** permet d'exprimer un critère de tri. **Elle peut être utilisée plusieurs fois pour des tris multi-critères.** Le poids d'un critère dépend de sa position relative sur la ligne de commande (du plus au moins important). Un critère est défini par **pos<sub>1</sub> [ , pos<sub>2</sub> ]**, avec la clé qui s'étend du champ numéro **pos<sub>1</sub>** jusqu'au numéro **pos<sub>2</sub>** compris. Si **pos<sub>2</sub>** n'est pas mentionné, la clé comprend toute la ligne à partir de **pos<sub>1</sub>**. Les champs sont numérotés à partir de 1.

Par défaut, la clé ainsi définie est traitée en tenant compte des options globales (**-r**, **-n**, etc.). Pour traiter une clé de façon particulière, on fait suivre les numéros des positions **pos<sub>1</sub>** et/ou **pos<sub>2</sub>** d'un ou plusieurs caractères parmi les suivants qui modifient le traitement du champ correspondant :

- **n** : traiter le champ comme un nombre
- **r** : inverser l'ordre de tri du champ
- **b** : ne pas tenir compte des blancs au début du champ
- **h** : traiter le champ comme une quantité humainement lisible
- **M** : traiter le champ comme une abréviation de mois
- et d'autres...

 Il y a d'autres possibilités offertes par **sort**. Consulter la documentation (**info** ou **man**).

## Exemples

```
$ sort -k 3,3n noms1 noms2
```

```
BON JEAN 13001
Patagueul James 13001
AMETTO LUCIE 13008
Aidubois Laure 13013
Golade Larry 13600
PEUPLU JEAN 13600
Epipiolli Jeff 83140
MAGNE CHARLES 84150
```

⇒ trie les fichiers *noms1* et *noms2* selon la valeur numérique du 3<sup>e</sup> champ (le code postal)

```
$ sort -k 3,3nr -k 2,2 noms1 noms2
```

```
MAGNE CHARLES 84150
Epipiolli Jeff 83140
PEUPLU JEAN 13600
Golade Larry 13600
Aidubois Laure 13013
AMETTO LUCIE 13008
Patagueul James 13001
BON JEAN 13001
```

⇨ trie les fichiers *noms1* et *noms2* selon l'ordre décroissant de la valeur numérique du 3<sup>e</sup> champ puis par ordre croissant du 2<sup>e</sup> champ (prénom)

```
$ sort -t : -k 3,3M fic
```

```
05:BB:janv.
0:c:JANV.
10:c:févr.
1:dd:mai
-3:eee:mai
1:AAA:juin
8:AAA:déc.
```

⇨ trie le fichier *fic*, où les champs sont délimités par :, selon le 3<sup>e</sup> champ contenant des mois abrégés

```
$ sort -t : -k 3,3M -k 1,1n fic
```

```
0:c:JANV.
05:BB:janv.
10:c:févr.
-3:eee:mai
1:dd:mai
1:AAA:juin
8:AAA:déc.
```

⇨ idem avec comme critère secondaire l'ordre numérique croissant du premier champ.

□

## 6.B.6 uniq : éliminer des lignes successives identiques

### Synopsis

```
uniq [-cdiu] [fic_entrée [fic_sortie]]
```

Sans option ni argument, la commande externe **uniq** filtre les lignes lues sur l'entrée standard et n'écrit sur sa sortie que des lignes uniques. Dans le jargon de **uniq**, une ligne est dite unique si elle n'est pas immédiatement répétée. Autrement dit, **uniq** écrit sur la sortie standard les lignes lues, en ne retenant que la première ligne parmi plusieurs lignes successives identiques.

Si précisé, les lignes sont lues dans le fichier *fic\_entrée* et non l'entrée standard. À la suite de *fic\_entrée*, *fic\_sortie* demande d'écrire dans *fic\_sortie* plutôt que sur la sortie standard. Notons que *fic\_sortie* est écrasé. Notons que pour écrire la sortie sur *fic\_sortie* tout en lisant l'entrée standard, il faut écrire :

```
uniq - fic_sortie
```

ou utiliser une redirection de la sortie.

### Exemple

```
$ cat fic_in1
a
A
bbb
bbb
bbb
a
cc
cc
d
$ uniq fic_in1
a
A
bbb
a
cc
d
```

⇒ On peut remarquer que les lignes successives identiques *bbb* et *cc* ont été filtrées et n'apparaissent plus qu'une seule fois.

□

🔍 On peut aussi remarquer dans la ligne *a* est unique même si elle apparaît plusieurs fois, car ses occurrences ne se suivent pas et **uniq** ne trie pas le fichier d'entrée. C'est pourquoi **uniq** est souvent utilisée en combinaison avec **sort**.

Les options offrent quelques variantes. En voici un extrait :

- l'option **-i** ignore la casse ;
- l'option **-u** demande de n'écrire que les lignes déjà uniques en entrée ;
- l'option **-d** fait l'inverse et demande de n'écrire que les lignes qui non uniques (commençant une suite d'au moins deux lignes identiques) ;
- l'option **-c** écrit au début de chaque ligne, son nombre d'occurrences consécutives.

### Exemples

```
$ uniq -i fic_in1
a
bbb
a
cc
d
```

⇒ En ignorant la casse, la ligne *A* a aussi été filtrée.

```
$ uniq -u fic_in1
a
A
a
d
```

⇒ Les lignes *bbb* et *ccc* ne sont pas uniques et sont filtrées (mais pas les lignes *a* car elles ne suivent pas).

```
$ uniq -d fic_in1
bbb
cc
```

⇒ seules les lignes *bbb* et *cc* commencent une suite de lignes identiques

```
$ uniq -c fic_in1
1 a
1 A
3 bbb
1 a
2 cc
1 d
```

⇒ les lignes sont précédées de leur nombre d'occurrences successives.

□





# Chapitre 7

## Impressions sous Unix

---

### 7.A lpr : lancer une impression

**lpr** (*line printer*) est une commande externe permettant d'imprimer des fichiers.

#### Synopsis

```
lpr [-Pprinter] [-#n] {-o option} {référence}
```



**L'option -P*printer* doit être utilisée à chaque impression au Département Informatique !**

C'est une option particulière car *printer* doit être le nom Unix de l'imprimante choisie. Dans la configuration actuelle au Département Informatique, *printer* doit être l'une des cinq imprimantes suivantes : **A**, **B**, **C**, **D** ou **E** selon la salle dans laquelle vous vous trouvez.

**lpr** lance le travail d'impression des fichiers référence ou, à défaut, de ce qui lu sur l'entrée standard. Si l'option **-#*n*** est spécifiée, l'impression de chaque fichier sera lancée *n* fois.

L'option **-o** permet de spécifier des options d'impression qui dépendent du système d'impression utilisé. Le système d'impression utilisé au Département d'Informatique s'appelle CUPS (*Common Unix Printing System*). Il propose notamment les options suivantes :

**job-sheets=standard** demande l'impression d'une page de garde qui contient le nom de l'utilisateur ayant demandé l'impression ainsi que le nom du fichier imprimé ;

**landscape** demande l'impression en orientation paysage ;

**prettyprint** demande une impression améliorée du document en ajoutant des numéros de page, le nom de l'utilisateur, etc. Si le document est le code source d'un programme écrit dans un langage reconnu (comme le C ou le C++), met en gras les mots-clés, en italique les commentaires, etc. ;

**number-up=*x*** demande de faire figurer *x* pages par feuille, où *x* peut prendre pour valeur 1, 2, 4, 6, 9 ou 16.

D'autres options sont possibles. La documentation de CUPS est accessible en Intranet via l'URL <http://allegro:631> (ou <http://localhost:631> sur la Debian).



*référence doit être une référence à un fichier imprimable, c'est à dire un fichier texte (y compris PostScript) ou PDF. On ne pourra pas imprimer ainsi un fichier doc par exemple.*

**lpr** ne réalise pas l'impression des fichiers *référence* immédiatement. Elle confie ce travail à un gestionnaire d'impressions (appelé aussi *spooler*) chargé de maintenir des files d'attente (appelées *spool*) pour l'accès aux imprimantes. Lorsqu'une imprimante se libère, le travail d'impression placé en tête de la file qui lui est associée est alors effectivement opéré.



Ne pas oublier que le matériel est relié en réseau et que l'on peut imprimer sur n'importe quelle imprimante des salles TP à partir des ordinateurs. D'où la nécessité de spécifier l'imprimante correspondant à la salle que vous occupez.

## Exemples

```
$ lpr -PA -o job-sheets=standard monfic.txt
```



*lancement de l'impression d'une page de garde suivie du fichier `monfic.txt` sur l'imprimante de la salle A.*

```
$ lpr -PA -o prettyprint -o number-up=2 source.cxx
```



*lancement de l'impression bien présentée d'un fichier C++ à raison de 2 pages par feuille sur l'imprimante de la salle A*

```
$ ls -l *.txt | lpr -PC
```



*lance l'impression sur l'imprimante C de la liste détaillée des fichiers du répertoire courant portant l'extension `txt`.*



## 7.B lpstat : obtenir des informations sur les imprimantes

La commande externe **lpstat** permet d'obtenir des informations sur les imprimantes installées et l'état du système d'impression.

### Synopsis

```
lpstat [-r] [-d] {-a [printer]} {-p [printer]}
```

L'option **-r** demande d'afficher si le serveur d'impression (pour nous, CUPS) est actif. L'option **-d** demande l'affichage du nom de l'imprimante par défaut. L'option **-a** demande de vérifier les files d'impression de l'imprimante spécifiée. Enfin, l'option **-p** demande de vérifier l'état de l'imprimante spécifiée.



Il existe d'autres commandes pour vérifier et/ou gérer les imprimantes parmi lesquelles **lpoptions** qui permet de vérifier et fixer des options pour des imprimantes et de définir une imprimante (file d'impression) par défaut, et **lpadmin** qui permet d'administrer les imprimantes.

## 7.C `lpq` : obtenir l'état d'une file d'attente d'impression

La commande externe `lpq` (*line printer queue*) permet d'obtenir des informations sur les (travaux d')impressions en attente.

### *Synopsis*

`lpq` [-**P***printer*]

Affiche sur sa sortie standard, l'état de la file d'attente pour l'imprimante *printer* si l'option **-P** est spécifiée. De nombreuses informations sont affichées concernant les impressions en attente, parmi lesquelles le nom de l'utilisateur ayant lancé l'impression et l'information de la colonne *Job-id* qui est très importante car elle va permettre de supprimer l'impression correspondante.



Il faut absolument utiliser l'option **-P** pour les mêmes raisons que `lpr`.

## 7.D `lprm` : supprimer des impressions en attente

Lorsqu'une impression est en attente, on peut la supprimer avec la commande externe `lprm`.

### *Synopsis*

`lprm` [-**P***printer*] *job-id* {*job-id*}

Supprime les impressions en attente pour l'imprimante *printer* (si **-P** est spécifiée) portant les numéros passés en arguments. Ces numéros peuvent être obtenus avec `lpq` et doivent correspondre à des impressions que l'utilisateur possède (à moins d'avoir des privilèges particuliers).



## Chapitre 8

# Courrier électronique

---

Le courrier électronique (ou *courriel* ou *mél* ou *mail* en anglais) est un moyen de communication permettant d'envoyer des messages à partir d'un ordinateur, à destination d'un utilisateur d'Internet. Il est utilisé au Département Informatique par les professeurs pour informer les étudiants, mais aussi par le secrétariat, l'association des étudiants...

- ❗ Votre adresse de messagerie officielle à l'Université (et donc au Département) a normalement la forme suivante :

*Prénom.Nom@etu.univ-amu.fr*

Les messages envoyés à cette adresse doivent être consultés quotidiennement, du moins les jours ouvrés, afin de vous tenir informés (ce qui relève de votre responsabilité). L'Université met à votre disposition un *Espace Numérique de Travail* (ENT) comprenant notamment un serveur Webmail vous permettant de consulter les messages envoyés à cette adresse, et d'en envoyer. Cet ENT est accessible via l'URL <http://ent.univ-amu.fr> une fois la charte de l'Université validée (voir procédure à suivre sur le site).

### 8.A mail : la messagerie électronique traditionnelle d'Unix

Mettons de côté la messagerie de l'Université. Le courrier électronique existe sous Unix depuis les années 80 et est à l'origine du courrier électronique échangé sur Internet actuellement. Sur un système Unix, la commande traditionnelle pour consulter et envoyer des messages électroniques est la commande externe **mail**.

- 💣 **Les Debians sur les PC ne sont pas configurées pour envoyer des courriers électroniques à travers Internet. Ce qui suit est donc exclusivement destiné à être utilisé sur la machine allegro et encore, juste le temps de manipuler la commande mail en TP, car ensuite, cela ne vous sera plus possible.**

**mail** est un peu archaïque et assez mal adaptée (dans sa version de base) pour l'envoi et la réception de messages contenant des pièces jointes. Cependant, elle fait partie de la culture que doit avoir un informaticien car il est fréquent que l'on soit obligé de s'en servir, en particulier pour l'écriture de scripts (programmes) shell.

❗ Sous Unix, il existe plusieurs logiciels (appelés *mailer*) pour le courrier électronique, dont **pine** qui est plus convivial que **mail**, et qui est adapté à la gestion des pièces jointes. Il y a aussi **thunderbird** (nommé **icedove** sous Debian) qui est un *mailer* graphique encore plus convivial.

Pour envoyer un courrier électronique à une personne, il faut qu'elle dispose d'une adresse e-mail. Outre votre adresse officielle à l'Université, vous disposez d'une adresse temporaire, propre au Département. Cette adresse correspond à une boîte à lettres sur allegro et **ne sera utile que pour les TP. Elle sera injoignable par la suite.**

L'adresse e-mail des étudiants au Département est composée de leur nom d'utilisateur (sur allegro) suivi de

**@allegro.iut.univ-aix.fr**

Par exemple, l'utilisateur et1001 a pour adresse **et1001@allegro.iut.univ-aix.fr**.

Lorsqu'un mail est reçu, il est stocké par le système dans la boîte aux lettres de(s) l'utilisateur(s) destinataire(s). Celui-ci peut consulter sa boîte à tout moment, et archiver ou détruire les messages reçus.

## 8.B Réexpédition des messages

Pour une raison ou une autre, certains utilisateurs préfèrent que les messages qui leur sont envoyés à une certaine adresse ne soient pas stockés sur le serveur de messagerie qui en a la charge mais qu'ils soient plutôt redirigés vers un autre serveur de messagerie, peut-être plus pratique à consulter.

Unix propose un moyen simple pour cela. Sur le serveur de messagerie, il faut créer le fichier `.forward` dans le répertoire d'accueil de l'utilisateur concerné, contenant l'adresse<sup>1</sup> électronique de la boîte vers laquelle les messages doivent être réexpédiés.

💣 **En contrepartie, les messages redirigés ne sont pas dupliqués. Ils ne seront stockés que sur le serveur cible de la redirection. Si l'on veut qu'ils le soient il faut ajouter une ligne avec votre nom d'utilisateur.**

✍️ Puisque la messagerie sur les PC et allegro sera désactivée, il ne sera pas utile de créer ce fichier (sauf éventuellement pour les besoins d'un TP).

### Exemple

Supposons que l'utilisateur et1001 sur allegro souhaite que tous les messages destinés à l'adresse `et1001@allegro.iut.univ-aix.fr` soient redirigés vers son adresse `paul.untel@etu.univ-amu.fr`, alors il doit créer le fichier `.forward` contenant cette adresse dans son répertoire d'accueil sur allegro.

□

## 8.C Consulter une boîte aux lettres électronique

**mail** permet de consulter les mails contenus dans une boîte aux lettres.

1. on peut mettre plusieurs adresses, à raison d'une par ligne

## Utilisation pour consultation

**mail** [-f référence]

Sans l'option **-f référence**, **mail** consulte la boîte aux lettres que le système a créée pour y stocker les messages reçus pour l'utilisateur en session. Sur allegro, la boîte de réception de tout utilisateur `etxxxx` est le fichier `/var/spool/mail/etxxxx`.



**Ce fichier devrait être vidé (à chaque consultation, en réalisant un archivage des messages) pour ne pas grossir exagérément, ce qui peut conduire à une incapacité de recevoir du courrier nouveau (voir la notion de quota dans la section 2.E.11 page 32), voire de travailler. Cela ne vous concernera pas car vous ne recevrez pas de messages sur allegro.**



Un fichier particulier, nommé `mbox`, est présent (ou sera créé à l'occasion du premier archivage) dans le répertoire de travail initial de l'utilisateur pour contenir les messages archivés.

Pour que **mail** affiche les messages contenus dans une autre boîte aux lettres que `/var/spool/mail/etxxxx`, il faut utiliser l'option **-f référence**, où *référence* est la référence de cette boîte (comme `~/mbox`). Les boîtes aux lettres ne sont autre que des fichiers texte contenant des mails. Elles peuvent être facilement créées par la sauvegarde ou l'archivage de messages avec la commande **mail**.

## Exemples

```
et1001$ mail
No mail for et1001
```

⇒ la boîte aux lettres de `et1001` (soit `/var/spool/mail/et1001`) ne contient aucun message (ancien ou nouveau).

```
cpb$ mail
Mail version 8.1 6/6/93. Type ? for help.
"/var/spool/mail/cpb": 2 messages 2 new
>N 1 cindy@crawford.net Sun Sep 2 16:22 16/416 "rendez-vous ce so"
  N 2 bidule@allegro.iut.univ- Sun Sep 2 16:34 17/428 "ne pas oublier..."
&
```

⇒ `cpb` a reçu 2 nouveaux messages. Ce sont les seuls messages contenus dans la boîte aux lettres `/var/spool/mail/cpb`.

```
$ mail -f mbox
Mail version 8.1 6/6/93. Type ? for help.
"/users/prof/cpb/mbox": 3 messages 0 new
> 1 didier.mathieu@univmed.f Mon Aug 26 18:37 18/536 "compte-rendu de l"
  2 didier.boitard@univmed.f Mon Aug 26 18:38 20/588 "Proposition exerc"
  3 marc.laporte@univmed.fr Mon Aug 26 19:25 14/461 "tu connais la der"
&
```

⇒ `cpb` possède un fichier `mbox` qui est une boîte aux lettres destinée à contenir les messages archivés. Ici, il y en a 3.



Lorsque **mail** affiche le contenu d'une boîte non vide, comme dans les deux dernières commandes, elle affiche en réalité une liste paginée des en-têtes (sorte d'extrait) des messages qu'elle contient. Chaque ligne commence

par le statut du message (**N** pour nouveau), l'expéditeur, la date d'expédition, sa taille (lignes/caractères) et le sujet.

**mail** attend ensuite que l'utilisateur tape une commande interne de **mail**, ce qui est indiqué par l'affichage du prompt **&** en dernière ligne (mode interactif). Le marqueur **>** indique sur quel message par défaut s'appliquera la commande tapée.

Il existe de nombreuses commandes internes de **mail** permettant de survoler la liste des messages, d'en sauver, d'en supprimer, d'en archiver ou d'y répondre. En voici un extrait (mêmes conventions que pour les commandes Unix) :

### 8.C.1 Obtenir de l'aide

**?** [*commande\_interne*]  affiche une aide sur la *commande\_interne* si elle est spécifiée ou un résumé des commandes sinon

### 8.C.2 Se déplacer dans la liste des en-têtes

**h** [*numéro*]  affiche la liste des en-têtes à partir du message numéro spécifié, ou la liste courante si aucun numéro n'est spécifié ;

**z**  affiche la page suivante de la liste des en-têtes ;

**z-**  affiche la page précédente de la liste des en-têtes ;

### 8.C.3 Lire un message

[*numéro*]  affiche le message *numéro* spécifié, ou le message précédé du marqueur **>** ;

**more** [*numéro*]  affiche en paginant le message *numéro* spécifié, ou le message précédé du marqueur ;

### 8.C.4 Répondre à un message

**R** [*numéro*]  permet de répondre à l'expéditeur du message *numéro*, ou à celui précédé du marqueur. La saisie du message est très sommaire et ne permet pas de revenir à une ligne précédente. La saisie prend fin lorsqu'une ligne est uniquement composée du caractère point ou lorsque **CTRL-D** est tapé (marque de fin de fichier). **mail** demande ensuite à qui doit-il envoyer des copies (carbon copy) de ce message (taper  si à personne) ;

**r** [*numéro*]  Idem que **R** mais le message est aussi envoyé à tous les destinataires du mail auquel on répond ;



**La commande **r** ne devrait être que très rarement utilisée, et à bon escient pour ne pas ennuyer les destinataires !!!**

### 8.C.5 Sauvegarder un message

**s** {<sub>l</sub>[*n*[-*m*]] *référence*  sauve les mails indiqués, à la fin du fichier *référence*. Cela n'a aucune incidence sur le contenu de la boîte aux lettres actuellement consultée ;



**w** {`[_[n[-m]]`} *référence* Entrée même chose que **s** mais seul le corps des messages est sauvé et pas l'en-tête<sup>2</sup>. S'il s'agit de la commande **mail** (voir section 8.E page suivante), sauve les pièces jointes séparément ;

### 8.C.6 Supprimer un message, annuler la suppression

**d** {`[_[n[-m]]`} Entrée marque les mails indiqués à supprimer. Ils ne seront effectivement supprimés que lorsque l'archivage est réalisé (commande interne **q**). *n* (si spécifié) est le numéro du mail à supprimer (sinon, c'est le mail précédé du marqueur). S'il est suivi de *-m*, alors le marquage est effectué jusqu'au mail *m*. On peut aussi spécifier une liste de numéros ou d'intervalles en les séparant par un espace ;

**u** {`[_[n[-m]]`} Entrée annule la commande **d** sur les messages indiqués ou sur celui précédé du marqueur ;

### 8.C.7 Archiver les messages

**q** Entrée supprime les messages à supprimer et archive les messages lus dans le fichier `mbox` du répertoire de travail de l'utilisateur en session, ou dans le fichier *référence* si l'option **-f référence** a été spécifiée. Seuls les messages non lus restent dans la boîte `/var/spool/mail/etxxxx` (pour l'utilisateur `etxxxx`).

### 8.C.8 Quitter sans modifier la boîte aux lettres

**x** Entrée quitte **mail** en laissant inchangée la boîte aux lettres.

## 8.D Envoyer un mail

Aux commandes internes de mail décrites ci-dessus, il faut rajouter la commande suivante :

**m** *destinataire* {*destinataire*} Entrée permet d'envoyer un mail aux destinataires spécifiés. La saisie du message se fait de la même manière que pour les commandes **R** et **r**.

Il existe aussi une autre utilisation de la commande **mail**, permettant d'envoyer des messages sans consulter de boîte aux lettres.

#### Utilisation pour l'envoi

**mail** [-s *sujet*] [-c *liste-cc*] [-b *liste-bcc*] *destinataire* {*destinataire*}

L'option **-s** permet de spécifier un sujet sur la ligne de commandes. Elle est pratique car en cas de redirection de l'entrée standard, **mail** ne demande pas de sujet.



*sujet* doit être un seul mot. S'il s'agit d'une phrase, il faudra protéger les blancs.




La redirection est particulièrement utile lorsqu'on veut envoyer un mail conséquent, car dans ce cas, il vaut mieux le saisir dans un éditeur (comme **vi** ou **emacs**) et le sauver dans un fichier, puis l'envoyer en redirigeant l'entrée de **mail**.

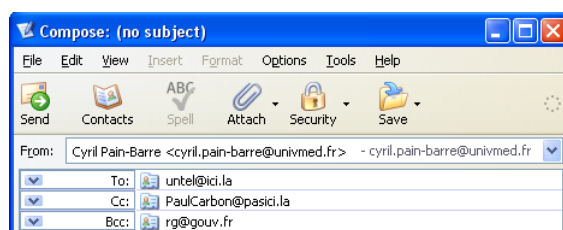
2. L'en-tête d'un mail est tout ce qui précède la première ligne vide du message. Le corps est tout ce qui la suit.

### 8.D.1 Carbon copies

On utilise les "*carbon copies*" lorsqu'on veut envoyer une copie du message à quelqu'un que ce message peut intéresser, alors qu'il n'en est pas le principal destinataire (par exemple, un message envoyé à un collaborateur peut intéresser d'autres personnes). On les utilise aussi pour s'envoyer une copie du message. L'option **-c** précise par *liste-cc* une liste de destinataires pour une "carbon copy", où *liste-cc* est une liste d'adresses mail séparées par des virgules.


 Les destinataires de la carbon copy sont indiqués dans l'entête du message par le champ **Cc :**, et sont connus des autres destinataires.

Notons que les gestionnaires de courrier électroniques permettent aussi de choisir le type de destinataire (**To :**, **Cc :**, **Bcc :**) :



### 8.D.2 Blind Carbon copies

Les "*blind carbon copies*" sont aussi des carbon copies, à la différence les autres destinataires n'ont pas connaissance de leur existence, ni de qui les a reçus (les destinataires des blind carbon copies ne sont indiqués dans aucun des messages reçus par les autres destinataires). L'option **-b** précise par *liste-bcc*, une liste de destinataires pour une "blind carbon copy".

 L'utilitaire **mail** permet bien d'autres choses. Notamment, l'inclusion de fichiers dans un mail en cours de rédaction, etc. (cf. le manuel en ligne).

## 8.E nail, une évolution de mail

Depuis quelques années, **mail** est généralement remplacé par **nail** (mais on l'invoque aussi par **mail**) qui lui ajoute des fonctionnalités, parmi lesquelles l'option :

{ **-a** *fichier-joint* }

Cette option demande de joindre un fichier au message, comme le font les gestionnaires de courrier évolués. On peut utiliser plusieurs fois **-a** pour joindre plusieurs fichiers.

D'autre part, lors de la consultation des messages, la commande interne **w** sauve les pièces jointes séparément.

# Chapitre 9

## Personnalisation de l'environnement bash

---

### 9.A Les alias

bash, comme d'autres interpréteurs, permet d'associer un nom (identificateur) à toute une ligne de commandes. Un tel identificateur est appelé un **alias**. S'il apparaît comme un nom de commande à exécuter, il sera remplacé par la ligne de commandes qu'il représente. Cette opération s'appelle l'expansion des alias et est réalisée avant les autres expansions (du tilde, des chemins, etc.).

#### Exemples

Supposons qu'il existe un alias **letd** qui représente la ligne de commandes **ls -l /home**, et observons le résultat de la commande suivante :

```
$ letd
total 141
drwx-----  7 abasop  abasop      1024 sep 12 13:41 abasop
drwx-----  6 abrnica abrnica      1024 sep 12 13:59 abrnica
drwx----- 10 albgui  albgui      1024 sep 12 14:00 albgui
...
```

⇒ **ls -l /home** a été exécutée à la place de **letd**.

```
$ letd > contenu
$
```

⇒ le fichier *contenu* est créé et contient ce que **ls -l /home** écrit sur sa sortie standard.

```
$ echo letd
letd
```

⇒ affiche le texte **letd**, car **letd** est placé en argument et n'est donc pas reconnu comme une commande, ni remplacé.

□

#### 9.A.1 alias : créer un alias d'une ligne de commandes

Pour créer un alias, on utilise la commande interne **alias**.

#### Synopsis

```
alias [ident [= 'ligne_commande' ]]
```

Sans argument, `alias` écrit la liste des alias créés. Avec seulement l'argument *ident*, elle donne la ligne de commandes qui a été associée à *ident*. Avec le `=`, elle associe *ident* à *ligne\_commande*.

✍ Notons que *ligne\_commande* n'est pas forcément réduite à une seule commande, mais peut être n'importe quelle suite de commandes valide (voir section 16.B.3 page 240). Notamment, ce peut être une séquence de commandes séparées par un `;` (point virgule).

### Exemples

```
toto$ alias
alias cp='cp -i'
alias ll='ls -l'
alias ls='ls --color=auto'
alias mv='mv -i'
alias rm='rm -i'
alias vi='vim'
toto$ alias ll
alias ll='ls -l'
toto$ alias unix='cd ~/tp/tpunix'
toto$ pwd
/home/toto
toto$ unix
toto$ pwd
/home/toto/tp/tpunix
```

□

❗ Certains alias indiqués par la première commande de l'exemple précédent sont des alias classiques, automatiquement créés lorsque vous vous logez (ouvrez une session) car ils sont contenus dans des fichiers que bash exécute lors du login d'un utilisateur.

### Exemple


À la rentrée 2008, le fichier `~/.bashrc` des utilisateurs sur les PC et allegro contenait entre autres les lignes suivantes :


```
# alias recommandés
alias ll='ls -l'
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'
alias vi='vim'

# some more ls aliases
#alias la='ls -A'
#alias l='ls -CF'
```

⇒ Le premier groupe de commandes est exécuté, alors que le second ne l'est pas car il est en **commentaires**. En supprimant le `#` devant les commandes **alias**, elles le seront.

□

 Contrairement aux shells csh et tcsh, il n'est pas possible dans bash de paramétrer les alias. Mais bash permet d'écrire des fonctions offrant des possibilités qui vont bien au delà des alias (t)csh.

 Un alias n'existe que dans le shell où il a été créé. Il disparaît quand le shell se termine. Pour créer un alias qui sera disponible depuis n'importe quel shell, il faudra placer sa création dans un fichier de configuration de bash (soit `.bash_profile` soit `.bashrc`). Il est toutefois conseillé de créer un fichier spécifique tel que `~/.bash_aliases` qui les contient et de s'arranger pour que ce fichier soit exécuté. Nous verrons cela dans la section [9.D](#) page [107](#).

## 9.A.2 Empêcher l'expansion d'un alias

Il arrive qu'on souhaite empêcher l'expansion d'un alias. Par exemple, l'alias **vi** vaut **vim** (**vim** est une version améliorée de l'éditeur de texte **vi**). Si l'on veut vraiment exécuter **vi** mais pas **vim**, il faut empêcher l'expansion de l'alias **vi**. Pour cela, il suffit d'utiliser une protection dans le mot **vi**.

### Exemple

Toutes les commandes suivantes empêchent l'expansion de l'alias **vi** et exécutent effectivement la commande **vi** :

```
untpau$ 'vi'
untpau$ 'v' i
untpau$ ""vi
untpau$ \vi
untpau$ v\i
...
```

□

## 9.A.3 unalias : supprimer un alias

Pour supprimer un alias dont on n'a plus besoin, il faut utiliser la commande interne **unalias**.

### Synopsis

```
unalias ident {ident}
```

Supprime les alias *ident* s'ils existent, sinon affiche une erreur.

### Exemples

```
toto$ alias unix
alias unix='cd ~/tp/tpunix'
toto$ unalias unix
toto$ unix
-bash: unix: command not found
toto$ unalias trucmuche
-bash: unalias: trucmuche: not found
```

□

## 9.B Introduction aux fonctions shell

Les fonctions seront détaillées un peu plus loin, en même temps que les scripts bash. Elles sont introduites ici afin de pouvoir contourner l'impossibilité de paramétrer les alias.

### Exemple

Il n'est pas possible de créer un alias qui serait nommé `llprint` et qui permettrait d'imprimer sur l'imprimante de la salle A les informations détaillées sur les fichiers ou répertoires passés en arguments. En effet, si l'on crée l'alias suivant :

```
$ alias llprint='ls -l | lpr -PA'
```

et qu'on l'invoque par :

```
$ llprint fic rep
```

alors, la commande qui sera exécutée est :

```
ls -l | lpr -PA fic rep
```

qui ne correspond pas à ce que l'on souhaite, c'est à dire :

```
ls -l fic rep | lpr -PA
```

□


À la place d'un alias, il faut créer une fonction. Deux syntaxes équivalentes permettent de créer une fonction de nom *nomfonction*.

### Synopsis de la création d'une fonction

<b>function</b> <i>nomfonction</i> {		<i>nomfonction</i> () {
<i>corps</i>	ou	<i>corps</i>
}		}

où **function** est un mot-clé de bash, et où le *corps* de la fonction **est constitué d'un nombre quelconque de lignes contenant des commandes**.

Une fonction est généralement destinée à recevoir des paramètres (arguments). Ceux-ci sont désignés dans le *corps* par les mots suivants, appelés **paramètres positionnels** : **\$1** pour le premier, **\$2** pour le second, ..., jusqu'à **\$9**. S'il y a plus de 9 arguments, on peut utiliser **\${n}** (par exemple **\${10}**). Lors de l'exécution de la fonction, si son corps contient des paramètres positionnels non protégés par des apostrophes ou le backslash, bash les remplacera par le ou les arguments correspondants.

 Les expressions commençant par **\$** ne sont pas protégées par les guillemets.

Enfin (et ce sera tout pour le moment), l'expression **\$\*** sera remplacée par la liste de tous les arguments, séparés par un blanc. D'autres manipulations seront vues en même temps que les variables et d'autres paramètres positionnels seront vus en même temps que les scripts bash.

### Exemple

Pour revenir à notre commande `llprint`, puisque nous ne connaissons pas par avance le nombre d'arguments, on utilisera l'expression **\$\*** :

```
$ function llprint {
> ls -l $* | lpr -PA
> }
$
```

⇒ création de la fonction `llprint`. Le `>` en début de ligne est le prompt de niveau 2. Il est affiché par `bash` pour indiquer que la ligne de commande n'est pas terminée (il attend le `}`).

```
$ llprint fic rep
```

⇒ exécute la commande souhaitée. En effet, le corps de `llprint` est exécuté où `$*` est remplacé par `fic rep`.

□

❗ À l'instar des alias, une fonction n'existe que dans le shell où elle a été créée. Elle disparaît quand le shell se termine. Pour créer une fonction qui sera disponible depuis n'importe quel shell, il faudra placer sa création dans un fichier de configuration de `bash` (soit `.bash_profile` soit `.bashrc`). Il est toutefois conseillé de créer un fichier spécifique tel que `~/bash_functions` qui les contient et de s'arranger pour que ce fichier soit exécuté. Nous verrons cela dans la section 9.D page 107.

## 9.C La variable d'environnement PATH

Les variables, et notamment les variables d'environnement, seront présentées en détail au chapitre 15. En attendant, il vous sera utile de modifier la variable **PATH** afin de l'adapter à vos besoins.

Ainsi que nous l'avons vu dans la section 3.A page 43, cette variable contient une liste de chemins consultés par `bash` pour rechercher les fichiers exécutables correspondant aux commandes externes tapées (et dont le premier mot ne comporte pas de `/`).

✍ Cette recherche n'est pas effectuée si le mot correspond à un alias, un mot-clé, une fonction ou une commande interne.

Dans le cas contraire, le fichier exécutable est recherché dans chacun des chemins de **PATH**, dans leur ordre d'apparition. Sur `allegro` et pour un utilisateur normal, cette variable contient probablement la chaîne suivante :

```
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

Les chemins étant séparés par `:`, ce *path* contient les 5 chemins suivants :

```
/usr/local/bin
/usr/bin
/bin
/usr/local/games
/usr/games
```


Ainsi le fichier exécutable sera d'abord recherché dans `/usr/local/bin`. S'il n'y est pas, la recherche se poursuit dans `/usr/bin`, etc. S'il n'est trouvé dans aucun chemin de **PATH**, `bash` affichera un message d'erreur.


Les chemins contenus dans le *path* précédent sont tous absolus mais le *path* peut contenir aussi des chemins relatifs. Dans ce cas, `bash` utilisera les chemins relatifs depuis le répertoire de travail lorsqu'il effectue la

recherche. Le chemin `.` (point) peut en faire partie, ce qui veut dire que bash recherchera aussi les fichiers exécutables dans le répertoire de travail. Le chemin vide aurait le même effet.

Voici quelques commandes permettant de modifier la variable **PATH** :

- vider son contenu :  
\$ **PATH=""**
- ajouter le chemin *chemin* au début :  
\$ **PATH="chemin:\$PATH"**
- ajouter le chemin *chemin* à la fin :  
\$ **PATH="\$PATH:chemin"**

 La modification de **PATH** n'affecte que le shell sur lequel elle a lieu. Lorsque celui-ci se termine, elle disparaît. Pour rendre la modification permanente, il faut placer les modifications dans l'un des fichiers de configuration de bash (normalement `~/.bash_profile` qui sera vu section 9.D.2.b page 109).

 **Il est très dangereux de modifier PATH et de placer le répertoire de travail avant les chemins vers les commandes externes courantes (/bin, /usr/bin, etc.). On s'expose alors à une attaque bien connue dans le monde Unix.**

## Exemples

Dans ces exemples, on suppose que l'étudiant bart est logé sur allegro :

```
bart:~/outils$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
```

⇒ le path semble normal

```
bart:~/outils$ pwd
/home/bart/outils
bart:~/outils$ ls -l pwd_bavard
-rwxr-xr-x  1 bart      simpson          70 nov 28 17:56 pwd_bavard
```

⇒ le répertoire de travail contient un exécutable nommé `pwd_bavard`

```
bart:~/outils$ pwd_bavard
pwd_bavard: Command not found.
```

⇒ mais il ne peut pas être exécuté si on tape seulement son nom car son chemin ne figure pas dans le path

```
bart:~/outils$ ./pwd_bavard
Cher ami, votre répertoire de travail est : /home/bart/outils
```

⇒ en revanche, si on utilise un `/` dans sa référence (le premier mot), bash ne tient pas compte du path et le trouve

```
bart:~/outils$ PATH="$PATH:."
```

⇒ on rajoute le répertoire de travail dans **PATH**



```
bart:~/outils$ pwd_bavard
```

Cher ami, votre répertoire de travail est : /home/bart/outils

⇒ *cette fois, on peut l'exécuter juste en tapant son nom*

```
bart:~/outils$ cd ..
```


```
bart:~$ pwd_bavard
```

```
-bash: pwd_bavard: command not found
```

⇒ *on n'est plus dans le répertoire contenant `pwd_bavard` donc bash ne le trouve pas dans le répertoire de travail*

```
bart:~$ PATH="$PATH:/home/bart/outils"
```

⇒ *on place le chemin absolu des répertoires contenant les outils (et `pwd_bavard`) dans le path.*

 *On aurait pu utiliser `~/outils` plutôt que `/home/bart/outils`*

```
bart:~$ pwd_bavard
```

Cher ami, votre répertoire de travail est : /home/bart

⇒ *`pwd_bavard` est retrouvé car il apparaît dans un des répertoires du path*




## 9.D Fichiers de personnalisation de bash

Que ce soit au démarrage ou à l'arrêt, bash exécute des fichiers de configuration (initialisation). Ce sont des fichiers texte contenant des lignes de commandes bash, appelés **scripts**. En les modifiant, l'utilisateur a la possibilité de personnaliser son environnement bash : ajout d'alias, de fonctions, modification du PATH...

### 9.D.1 Fichiers exécutés au démarrage de bash

Parmi les fichiers permettant de personnaliser bash lors de son démarrage, certains sont communs à tous les utilisateurs et d'autres sont propres à chaque utilisateur :

- fichiers communs à tous les utilisateurs (configuration globale) : **/etc/profile** et **/etc/bash.bashrc**. Ces fichiers fixent une configuration d'ordre général pour les utilisateurs. Ils ne peuvent être modifiés que par root ;
- fichiers propre à chaque utilisateur (configuration utilisateur) : **~/.bash\_profile** et **~/.bashrc**. Chaque utilisateur dispose<sup>1</sup> en principe de ces fichiers cachés dans son répertoire d'accueil. Il peut les modifier pour configurer bash selon ses propres besoins.

 Notons que si bash ne peut lire (utiliser) `~/.bash_profile`, il essaiera de lire le fichier `~/.bash_login` (plus rare), et en dernier recours `~/.profile` (aussi utilisé par le shell sh). Ceci pour des raisons historiques et de compatibilité. Dans ce qui suit, nous ne ferons référence qu'à `~/.bash_profile` mais cela pourrait être l'un des deux autres fichiers.

Cependant, tous ces fichiers ne sont pas exécutés. Cela dépend du **type de bash** qui démarre. On peut distinguer principalement 3 types :

1. Lors de la création des utilisateurs, leur répertoire d'accueil est créé comme une copie du répertoire `/etc/skel` qui est un squelette minimal contenant de tels fichiers (voir 11.A.2).

1. **le login shell** (ou **shell de connexion**, ou encore **shell d'ouverture de session**) : tire son nom d'une époque antique où l'environnement de travail n'était pas graphique, et l'écran n'affichait que du texte. L'utilisateur ne disposait que d'un seul terminal où il tapait ses commandes. Ce shell était celui exécuté lorsque l'utilisateur se logeait. Aujourd'hui, avec l'environnement graphique, l'ouverture de session donne lieu à l'affichage d'un **bureau** (GNUStep, Gnome, Kde, ...) où il n'est généralement plus exécuté par défaut. Il l'est toutefois pour les 6 terminaux texte créés lors du démarrage (les `tty1` à `tty6`, accessibles par les combinaisons `CRTL+ALT+F1` à `CRTL+ALT+F6`) ainsi que pour les connexions à distance en mode texte (via TELNET ou SSH). Ce type de shell reste très utile.
2. **le shell interactif** : shell qui affiche le prompt, analyse les lignes de commandes tapées par l'utilisateur et les exécute. C'est donc un shell semblable à celui des exemples de ce cours. On notera qu'un login-shell est la plupart du temps interactif.
3. **le shell non-interactif** : shell qui n'interagit pas avec l'utilisateur et n'affiche pas de prompt. Ce type de shell est créé pour exécuter des scripts bash (fichiers contenant des commandes).

Selon son type, bash n'exécute que certains fichiers :

- un **login shell (interactif ou non)** : exécute uniquement et dans cet ordre, les fichiers :

- ◊ `/etc/profile`
- ◊ `~/.bash_profile`

**i** la section **9.D.2.c** explique comment faire en sorte d'exécuter aussi les fichiers `/etc/bash.bashrc` et/ou `~/.bashrc`.

- un **shell interactif (hors login-shell)** exécute uniquement et dans cet ordre, les fichiers :

- ◊ `/etc/bash.bashrc`
- ◊ `~/.bashrc`

- un **shell non-interactif** n'exécute aucun fichier de démarrage. Néanmoins, si la variable d'environnement **BASH\_ENV** est définie, elle contient le chemin d'un fichier qui est exécuté.

## 9.D.2 Rôle des fichiers de démarrage

Les fichiers de démarrage contiennent des commandes à exécuter (ou plutôt des instructions, car bash est un langage de programmation), selon le shell qui est exécuté. Ils changent d'une distribution à l'autre, mais aussi entre différentes versions d'une même distribution. Les applications installées et l'environnement matériel/réseau peuvent aussi avoir leur influence sur ces fichiers. Cependant, ils ont un rôle bien défini.

### 9.D.2.a Le fichier `/etc/profile`

Comme la plupart des fichiers situés dans `/etc`, le fichier `/etc/profile` concerne une configuration globale. Il est commun à tous les utilisateurs et ne peut être modifié que par root. C'est le premier fichier exécuté par un login-shell. Il fixe un environnement minimal pour l'utilisateur : variable **PATH**, masque de création de fichiers, prompt, ...

Il crée et initialise la variable **PATH**, de manière différente pour root, avec une instruction de type :

```
if [ "`id -u`" -eq 0 ]; then
    PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
else
    PATH="/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games"
fi
```

⇒ en deux mots, pour root, il faut les chemins vers les outils d'administration (/usr/local/sbin, /usr/sbin et /sbin) mais pas pour les autres utilisateurs à qui il faut plutôt les chemins vers les jeux (/usr/local/games et /usr/games).

Elle est normalement suivie de la commande :

```
export PATH
```

"exportant" la variable **PATH** qui devient une variable d'environnement (voir 17) ; une copie en sera communiquée à tous les processus (commandes) qui seront exécutés (directement ou indirectement) par ce shell.

Il initialise le masque de création de fichiers avec la commande **umask**, généralement ainsi :

```
umask 022
```

❗ Différentes valeurs du masque sont toutefois possibles : 002, 027, 077... Aussi, selon la distribution, l'initialisation du masque n'apparaît plus dans ce fichier mais est confiée au module dédié **pam\_umask**, sollicité lors de la connexion d'un utilisateur.

### 9.D.2.b Le fichier ~/.bash\_profile

Le fichier ~/.bash\_profile (ou équivalent) est exécuté par les logins-shells après /etc/profile. De ce fait, ses instructions peuvent modifier l'environnement mis en place dans /etc/profile.

Chaque utilisateur possède sa propre version, généralement copiée depuis /etc/skel lors de la création de l'utilisateur. L'utilisateur y modifie son environnement selon ses propres besoins, notamment la variable **PATH**, le masque de création de fichiers, etc.

Par exemple, il contient souvent l'instruction :

```
# set PATH so it includes user's private bin if it exists
if [ -d ~/bin ] ; then
    PATH=~/bin:"${PATH}"
fi
```

pour ajouter en début de **PATH** le répertoire ~/bin de l'utilisateur, s'il existe, dans lequel l'utilisateur peut placer ses utilitaires.


### 9.D.2.c Extension des fichiers /etc/profile et ~/.bash\_profile

Les login-shells n'exécutent que les fichiers /etc/profile et ~/.bash\_profile. Mais quand ils sont interactifs, on peut souhaiter qu'ils exécutent aussi les instructions de /etc/bash.bashrc et/ou de ~/.bashrc. Pour cela, il suffit d'ajouter une instruction dans /etc/profile et/ou ~/.bash\_profile.

Un utilisateur ne peut que modifier `~/.bash_profile` s'il veut que ses login-shells exécutent aussi `~/.bashrc`, et y ajouter une instruction telle que :

```
# include .bashrc if it exists
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

qui exécute les instructions de `~/.bashrc` s'il existe. L'inclusion de `/etc/bash.bashrc` se fait de la même manière.

 Cette instruction (ou presque) est parfois déjà présente par défaut dans `~/.bash_profile`.

#### 9.D.2.d Le fichier `/etc/bash.bashrc`

C'est le premier fichier exécuté par un shell interactif (hors login-shell). Il est commun à tous les utilisateurs et ne peut être modifié que par root. Il active quelques facilités et initialise l'invite de commandes.

#### 9.D.2.e Le fichier `~/.bashrc`

Il est exécuté à la suite de `/etc/bash.bashrc` par les shells interactifs (hors login-shell). Ses instructions peuvent donc modifier la configuration réalisée par `/etc/bash.bashrc`. Chaque utilisateur possède sa propre version, copiée depuis `/etc/skel` lors de la création d'un utilisateur.

L'utilisateur peut personnaliser ses shells interactifs en modifiant ce fichier. Le plus souvent, cette modification concerne la création d'alias et de fonctions. Notamment, ce fichier contient souvent la définition de certains alias classiques (voir 9.A.1).

L'utilisateur peut y ajouter ses propres alias et fonctions, mais il est préférable de les placer dans un autre fichier. Dans ce sens, `~/.bashrc` contient souvent une instruction telle que :

```
if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi
```

encourageant les utilisateurs à placer leurs propres alias dans le fichier `~/.bash_aliases`.

On peut y ajouter des lignes similaires pour exécuter le fichier `~/.bash_functions` qui contiendrait des définitions de fonctions.

### 9.D.3 Comment rendre permanente la personnalisation de son environnement ?

Nous avons vu que la création des alias, des fonctions et la modification de **PATH** n'était pas permanente. Pour qu'elle le soit, il faut modifier les fichiers `~/.bash_profile` et/ou `~/.bashrc`.

#### 9.D.3.a Où placer les modifications de la variable **PATH** ?

Cette variable, comme toutes les variables d'environnement, devrait être créée/modifiée dans les fichiers `/etc/profile` et `~/.bash_profile`. C'est donc dans `~/.bash_profile` que l'utilisateur doit la personnaliser.

### 9.D.3.b Où placer la création des alias et des fonctions ?

Les alias et les fonctions sont destinées à être utilisés dans des shells interactifs. Ils doivent être créés dans les fichiers `/etc/bash.bashrc` et `~/.bashrc`. L'utilisateur pourra ajouter les siens dans `~/.bashrc` mais il est préférable de placer les alias dans un fichier `.bash_aliases` et les fonctions dans `.bash_functions` et les faire exécuter dans `~/.bashrc`.

### 9.D.3.c Où placer la définition du masque des permissions avec umask ?

En principe, la commande **umask** ne devrait être exécutée que par les login shells, à partir desquels les autres commandes sont exécutées. Celles-ci héritent alors du masque défini (comme elles héritent du PATH). Ainsi, le masque **devrait** être défini dans les fichiers `/etc/profile` et `~/.bash_profile`. C'est donc dans `~/.bash_profile` que l'utilisateur devrait le personnaliser.

## 9.D.4 Particularité de l'environnement graphique

Ainsi qu'il a été dit précédemment, dans l'environnement graphique l'ouverture de session donne lieu à l'affichage d'un bureau et il n'y a pas de login shell d'exécuté. Cependant, lorsqu'on ouvre une fenêtre terminal (comme **gnome-terminal**, ou **aterm**, **rxvt**, ...), où un shell sera exécuté, il est possible de faire en sorte que ce dernier soit un login shell.

❶ Les terminaux graphiques peuvent être lancés à partir d'un shell en utilisant la commande correspondante (**aterm**, **gnome-terminal**, etc.). Généralement, ces commandes admettent une option pour que le terminal exécute un login-shell. Pour **aterm**, cette option est **-ls**. Il n'y en a pas pour **gnome-terminal**, pour lequel il faut passer par des profils. **gnome-terminal** est l'application terminal native du bureau **Gnome**, installé par défaut sur Debian. Pour que ce terminal lance l'exécution d'un login shell, il faut modifier son **profil** depuis son menu «Édition». On choisit alors d'éditer le **profil par défaut** et, dans l'onglet «Titre et commande», il faut cocher la case «Lancer la commande en tant que shell de connexion» (voir figure 9.1).

## 9.D.5 Fichier exécuté à l'arrêt d'un login shell

Lorsqu'un login shell se termine, il exécute le fichier `~/.bash_logout` si celui-ci existe. Du temps des écrans en mode texte, ce fichier contenait généralement la commande externe **clear** qui efface l'écran. De nos jours, peu de gens utilisent cette fonctionnalité.

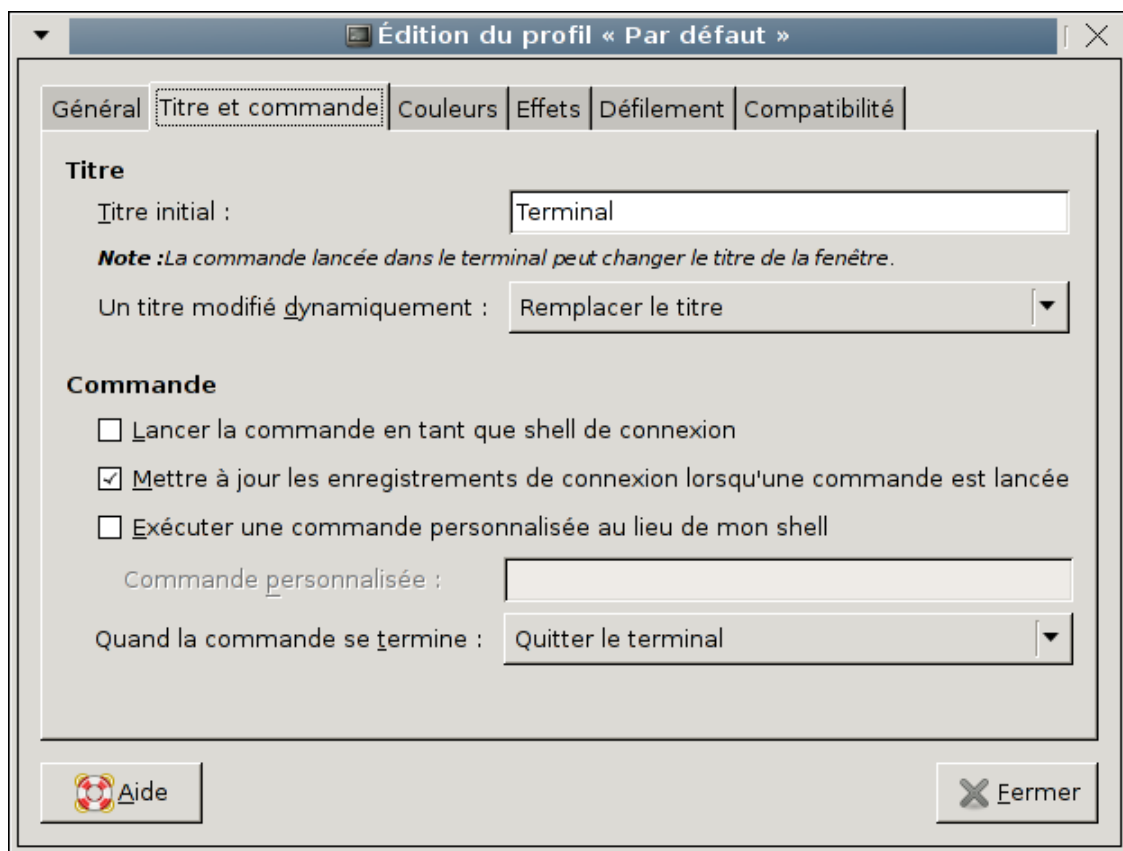


FIGURE 9.1 – Édition d'un profil de gnome-terminal. Il faut cocher la case adéquate pour que le terminal exécute un login-shell.

# Chapitre 10

## Les processus

---

### 10.A Introduction

Pour rester très simple, un processus est ce que le système crée afin d'exécuter un programme. Il possède un environnement qui comporte le code exécutable, la mémoire allouée par le système, des fichiers ouverts en lecture et/ou écriture, la pile d'exécution. . .

La différence entre un programme et un processus peut se comprendre facilement, en remarquant qu'un même programme comme **bash** (le fichier binaire exécutable `/bin/bash`) peut être exécuté par plusieurs utilisateurs sur plusieurs fenêtres (ou sur la même) en même temps. C'est un processus distinct qui est créé à chaque fois alors que c'est le même programme. Ces processus sont indépendants et disposent chacun d'une partie de la mémoire, de leurs propres entrées-sorties, etc.

Un processus est identifié par un numéro unique : le **PID** (*processus identifier*). Il a un propriétaire et (au moins) un groupe, qui sont utilisés par le système pour autoriser ou interdire les opérations qu'il effectue (lecture/écriture dans un fichier, dans la mémoire, . . .). Par exemple, lorsqu'un utilisateur lance la commande **ls**, le processus créé pour l'exécuter est propriété de l'utilisateur et possède son groupe<sup>1</sup>. Lorsque ce processus tente de lire le contenu d'un répertoire, le système utilise ces informations (et celles du répertoire) pour vérifier qu'il a bien le droit de le faire.

### 10.B Vie d'un processus

Un processus est toujours créé (lancé) par un autre, appelé son **processus père** (*parent processus*). Exception faite du processus **init**, portant le numéro 1, qui est le patriarche de tous car c'est le noyau (le cœur) du système Unix. Un processus a accès au numéro de processus de son père, appelé le **PPID** (*parent pid*). De même, lorsqu'un processus crée un **processus fils**, il en récupère le **PID**.


Durant sa vie, un processus passe par différents **états**, notamment d'exécution (par le processeur) et d'attente (d'une ressource). Il finit généralement par mourir, notamment lorsqu'il a terminé l'exécution de son code (programme). Le système permet à son père d'avoir connaissance de cet événement. Il revient alors à son père de le détruire effectivement. Un processus mort est en état de **zombie** tant que son père ne l'a pas effectivement détruit. Au cas où un processus meurt avant son fils (!), la paternité du fils revient à **init**.

---

1. Unix permet toutefois d'exécuter des programmes avec une autre identité.

### 10.B.1 Lancement par le shell d'un processus en premier plan

Lorsqu'on entre une commande externe, correspondant à un fichier exécutable, le shell (pour nous, bash) va demander au système de créer un processus fils pour exécuter cette commande. Puis il attend que son fils *meure* (parce qu'il a fini son office ou qu'il a été interrompu) avant d'afficher le *prompt* invitant à entrer une nouvelle commande.

 Lorsqu'elles ne figurent pas dans un *pipeline*, les commandes internes (comme **cd**, **pwd**, **umask**, **alias**...) sont elles exécutées directement par bash, sans créer de processus.

Ce comportement dit **séquentiel** ou **synchrone** est exactement celui que l'on attend généralement, car on n'aimerait pas voir le *prompt* s'afficher en plein milieu de ce qu'affiche le processus de notre commande, et ne plus savoir ensuite si il est terminé. Il est aussi nécessaire car la commande peut avoir besoin de lire des entrées depuis le clavier, et serait en compétition avec bash qui attendrait une commande après avoir affiché le prompt.

 On dit que le processus est exécuté en **premier plan** (*foreground*).

#### Exemples

```
$ ls -l /bin
```

```
# bash exécute ls et attend qu'elle se termine...
```

```
total 5236
```

```
-rwxr-xr-x 1 root root 811156 10 avril 2010 bash
-rwxr-xr-x 3 root root 26356 21 sept. 2010 bunzip2
-rwxr-xr-x 1 root root 408588 15 nov. 2010 busybox
-rwxr-xr-x 3 root root 26356 21 sept. 2010 bzip2
lrwxrwxrwx 1 root root 6 24 sept. 2010 bzip2 -> bzip2diff
-rwxr-xr-x 1 root root 2140 21 sept. 2010 bzip2diff
...
```

```
# ...avant de reprendre la main et d'afficher le prompt
```

```
$ vi fic
```

```
# c'est aussi le cas pour vi qui, contrairement à ls,
# a besoin de lire les entrées du clavier
```

□

Les séquences de commandes, par exemple *commande<sub>1</sub>* ; *commande<sub>2</sub>*, sont aussi exécutées séquentiellement : bash attend d'abord la fin de *commande<sub>1</sub>* puis exécute *commande<sub>2</sub>* et attend sa fin avant d'afficher le prompt.

Cependant, il y a des processus dont on ne veut pas attendre la fin pour continuer à taper des commandes. C'est le cas par exemple des applications graphiques (qui ouvrent au moins une fenêtre pour dialoguer), ou des programmes n'interagissant pas avec les utilisateurs, notamment les démons<sup>2</sup> (*daemon*).

### 10.B.2 Lancement par le shell d'un processus en tâche de fond


Afin d'éviter cette attente, on peut très simplement lancer des commandes en **tâches de fond**, ou **arrière-plan** (*background*), qui s'exécutent en parallèle. Dans ce cas, le shell n'attendra pas la mort du processus pour conti-

2. Un démon est un programme rendant un service, généralement exécuté en arrière-plan. Des mécanismes de communication permettent aux autres processus de le contacter afin de bénéficier de ses services. Par exemple, le gestionnaire d'impression **cupsd** est exécuté au démarrage pour satisfaire les requêtes d'impression.



nuer son traitement, c'est à dire afficher le *prompt* et attendre une nouvelle commande.

Le lancement en tâche de fond (*background*) se demande sur la ligne de commande de bash en rajoutant le caractère spécial **&** (*esperluette* ou *et-commercial*) tout à la fin de l'expression de la commande (après les options, les arguments et les redirections).

 Le caractère **&** est un séparateur de commandes.

## Exemples

```
$ aterm &
```

```
[1] 4767
```

⇒ *lancement en parallèle d'une fenêtre **aterm**. Le processus créé porte le numéro 4767 et son numéro de job est 1 (voir commande **jobs** section 10.C.1 page suivante) ;*

```
$ cmd1 -o1 arg1 &
```

```
[2] 4801
```

⇒ *lancement en parallèle de **cmd1** avec l'option **-o1** et l'argument **arg1**. Le processus créé porte le numéro 4801 (job numéro 2).*

```
$ cmd1 -o1 arg1 < ref1 &
```

```
[2] 4801
```

⇒ *comme précédemment mais avec en plus la redirection de l'entrée standard depuis le fichier **ref1** pour ce processus.*

```
$ cmd1 -o1 arg1 < ref1 & cmd2 -o2 arg2
```

⇒ *a pour effet de lancer en parallèle la commande **cmd1 -o1 arg1 < ref1** et de lancer en séquentiel (en attendant sa fin), la commande **cmd2 -o2 arg2**.*


```
$ cmd1 -o1 arg1 < ref1 & cmd2 -o2 arg2 &
```

⇒ *cette fois la deuxième commande est aussi affichée en parallèle*

□



**En contrepartie du lancement parallèle, la commande ne peut rien recevoir du clavier sur la fenêtre d'où bash l'a lancé puisque c'est bash qui va en avoir besoin pour recevoir une autre commande de la part de l'utilisateur.**

 Les applications graphiques quant à elles, ouvrent au moins une fenêtre et demandent au système un nouveau terminal (donc un nouveau fichier) pour utiliser le clavier dans cette fenêtre. C'est notamment le cas des consoles graphiques de type **aterm**, **rxvt**, etc. qui vont créer un nouveau terminal qui sera utilisé par le shell de la console.

## 10.C État des processus

### 10.C.1 jobs : lister les processus lancés par le shell

La commande interne **jobs** demande à bash l’affichage de la liste des processus fils qu’il a lancé, et qui ne sont pas terminés. Le numéro de job est celui affiché entre crochets lorsqu’on lance un processus en parallèle (voir exemples précédents).

#### Synopsis

```
jobs [-l]
```

L’option **-l** demande aussi l’affichage du numéro de processus.


#### Exemples

```
$ atrm &
[1] 3626
$ find / -name truc > machin.txt &
[2] 3673
$ jobs
[1] - Running          atrm
[2] + Running          find / -name truc > machin.txt
$ jobs -l
[1] - 3626   Running    atrm
[2] + 3673   Running    find / -name truc > machin.txt
```

□

### 10.C.2 ps : obtenir une liste de processus existants

La commande externe **ps** permet d’obtenir une liste de tout ou partie des processus créés par le système et non encore détruits (en activité ou zombies). Elle admet de très nombreuses options (en particulier sous Linux) dont seulement celles indiquées ici vous serviront habituellement. Comme d’habitude, se reporter au manuel en ligne (man) pour connaître toutes les options.

 Il existe de nombreuses syntaxes d’utilisation (options Unix, options BSD, options GNU). Nous choisissons ici la syntaxe BSD. De plus, cette commande ne se comporte pas tout à fait de la même manière selon les distributions (notamment entre la Debian et la Mandriva). Il est possible que la description ci-dessous ne corresponde pas tout à fait à votre installation.

#### Synopsis

```
ps [x] [a] [f] [l|u] {w}
```



**Attention, il n’y a pas de tiret devant les options (syntaxe BSD).**

Sans option, **ps** affiche simplement les informations sur les processus de l’utilisateur courant, qui sont rattachés au terminal (fenêtre) sur lequel elle a été lancée. Pour chaque processus, les informations suivantes sont affichées :

- PID : le numéro du processus ;
- TTY : le (fichier) terminal associé au processus, en principe contenu dans /dev. Le caractère ? marque l'absence de terminal ;
- TIME : le temps CPU du processus ;
- CMD : le nom de l'exécutable.

### Exemples

```
$ ps
  PID TTY          TIME CMD
 4722 pts/8        00:00:00 bash
 4790 pts/8        00:00:00 ps
```

➡ Les seuls processus actifs sur la fenêtre sont *bash*<sup>3</sup> et *ps* (qui va mourir immédiatement après avoir écrit cette liste). En rajoutant /dev/ devant l'élément TTY, on obtient la référence absolue du (fichier) terminal auquel le processus est rattaché.

```
$ aterm &
[1] 4767
```

➡ Lancement en parallèle d'une console *aterm* (application graphique simulant un terminal).

```
$ ps
  PID TTY          TIME CMD
 4722 pts/8        00:00:00 bash
 4767 pts/8        00:00:00 aterm
 4790 pts/8        00:00:00 ps
```

➡ Le processus *aterm* porte le numéro 4767 et est rattaché au même terminal. Cet *aterm* a normalement créé un processus fils pour lancer un shell *bash*. En revanche, ce nouveau *bash* dispose d'un nouveau terminal auquel il est rattaché.

□

L'utilisation d'options BSD (les seules présentées ici) ajoute la colonne STAT et remplace la colonne CMD par la colonne COMMAND qui donne plus d'informations que CMD (notamment les arguments de la commande). La colonne STAT indique l'état du processus qui peut être :

- D : dormant non interruptible ;
- R : s'exécutant ou pouvant d'exécuter ;
- S : dormant mais interruptible ;
- T : suspendu ;
- Z : zombie.

Des caractères additionnels peuvent figurer pour donner des informations complémentaires.

### Options de sélection de processus

Avec l'option **x**, tous les processus de l'utilisateur sont affichés même ceux qui ne sont pas rattachés au terminal. L'option **a** (*all*) demande aussi l'affichage des processus n'appartenant pas à l'utilisateur mais rattachés à un terminal. La combinaison **ax** demande l'affichage de tous les processus.

### Exemples

3. Le nom qu'il porte peut être différent : ce peut être aussi *-bash*, *sh* ou *-sh*

\$ **ps x**

PID	TTY	STAT	TIME	COMMAND
4696	?	S	0:00	/usr/X11R6/bin/wmaker
4720	?	R	0:00	aterm -ls
4722	pts/8	S	0:00	-bash
4767	pts/8	S	0:00	aterm
4768	pts/10	S	0:00	bash
4792	pts/8	R	0:00	ps x

⇒ On remarque que l'utilisation d'une option fait afficher les colonnes *STAT* et *COMMAND*. Avec l'option **x**, tous les processus de l'utilisateur apparaissent. Les deux premiers ne sont rattachés à aucun terminal. Le **bash** (pid 4768) qui a été créé par le **aterm** précédent dispose d'un nouveau terminal qui correspond au fichier */dev/pts/10*.

\$ **ps a**

PID	TTY	STAT	TIME	COMMAND
4468	tty7	Ss+	36:27	/usr/X11R6/bin/X :0 -audit 0 -auth /var/lib/gdm/:0.Xa
4683	tty1	Ss+	0:00	/sbin/getty 38400 tty1
4684	tty2	Ss+	0:00	/sbin/getty 38400 tty2
4685	tty3	Ss+	0:00	/sbin/getty 38400 tty3
4686	tty4	Ss+	0:00	/sbin/getty 38400 tty4
4687	tty5	Ss+	0:00	/sbin/getty 38400 tty5
4688	tty6	Ss+	0:00	/sbin/getty 38400 tty6
4722	pts/8	S	0:00	-bash
4767	pts/8	S	0:00	aterm
4768	pts/10	S	0:00	bash
4850	pts/1	Ss	0:00	-bash
4890	pts/1	S+	0:00	more CSetud1
4892	pts/8	R	0:00	ps a

⇒ L'option **a** fait afficher tous les processus rattachés à un terminal, quel que soit l'utilisateur.

\$ **ps ax**

PID	TTY	STAT	TIME	COMMAND
1	?	Ss	0:08	init [2]
2	?	S<	0:00	[kthreadd]
3	?	S<	0:00	[migration/0]
4	?	S<	8:14	[ksoftirqd/0]
5	?	S<	0:01	[watchdog/0]
...	...	...	...	...
3505	?	Ss	0:23	/sbin/syslogd
3514	?	Ss	0:00	/sbin/klogd -x
3525	?	Ss	0:01	/usr/bin/dbus-daemon --system
3554	?	Ss	0:00	/usr/sbin/sshd
...	...	...	...	...
4468	tty7	Ss+	36:27	/usr/X11R6/bin/X :0 -audit 0 -auth /var/lib/gdm/:0.Xa
4683	tty1	Ss+	0:00	/sbin/getty 38400 tty1
4684	tty2	Ss+	0:00	/sbin/getty 38400 tty2
4685	tty3	Ss+	0:00	/sbin/getty 38400 tty3
4686	tty4	Ss+	0:00	/sbin/getty 38400 tty4
4687	tty5	Ss+	0:00	/sbin/getty 38400 tty5
4688	tty6	Ss+	0:00	/sbin/getty 38400 tty6
...	...	...	...	...
4720	?	R	0:00	aterm -ls
4722	pts/8	S	0:00	-bash

```

4767 pts/8      S          0:00 aterm
4768 pts/10     S          0:00 bash
4849 ?          S          0:00 sshd: leborgne@pts/1
4850 pts/1       Ss         0:00 -bash
5892 pts/8      R          0:00 ps ax

```

☞ La combinaison **ax** fait afficher tous les processus, parmi lesquels *init* (pid 1).



❗ Les processus dont les noms apparaissent entre crochets (comme **kthreadd** et **migration** dans l'exemple précédent) font partie du noyau.

## Options d'affichage

✍ À noter que les options suivantes, en plus de modifier l'affichage, provoquent aussi l'affichage des processus de l'utilisateur s'ils sont rattachés à un terminal.

L'option **f** (*forêt*) demande d'utiliser des caractères ASCII dans le nom de la commande afin de mettre en évidence la paternité des processus (généalogie).

Les options **l** et **u** sont **exclusives**. L'option **l** (*long*) demande un affichage d'informations comprenant 13 colonnes :

```
F  UID  PID  PPID  PRI  NI  VSZ  RSS  WCHAN  STAT  TTY  TIME  COMMAND
```

où :

- **F** : (flags) affiche un indicateur sur le processus (hors de propos)
- **UID** : identifiant utilisateur (effectif)
- **PPID** : pid du processus père
- **PRI** : priorité du processus (plus sa valeur est grande, moins le processus est prioritaire). Ce nombre est la somme de la priorité de départ du processus et de la valeur de politesse associée
- **NI** : valeur de politesse comprise entre -20 et 19. Plus elle est grande, moins le processus est prioritaire ;
- **VSZ** : taille de la mémoire virtuelle du processus en Ko
- **RSS** : taille de la mémoire résidente (non transférée en *swap*)
- **WCHAN** : nom de la fonction du noyau dont le processus (en sommeil) attend le retour. Affiche **-** ou **\*** si le processus n'est pas en sommeil.

L'option **u** (*user*) demande un affichage orienté utilisateur et comprenant 11 colonnes :

```
USER  PID  %CPU  %MEM  VSZ  RSS  TTY  STAT  START  TIME  COMMAND
```

où :

- **USER** : nom d'utilisateur (effectif)
- **%CPU** : pourcentage d'utilisation de la CPU (ratio temps réel/temps d'utilisation de la CPU)
- **%MEM** : rapport entre la mémoire résidente et la mémoire physique de la machine
- **START** : heure ou date de démarrage d'un processus. L'affichage dépend de son âge : uniquement année ou mois et jour ou heure et minute.

Enfin, l'option **w** (*wide*) demande un affichage large. Normalement, il n'y a qu'une ligne par processus, limitée à 80 caractères, ce qui empêche parfois l'écriture complète de `COMMAND`. L'utilisation de **w** permet d'aller au delà de 80 caractères. En doublant le **w**, il n'y a pas de limite au nombre de caractères utilisés.

**i** Il existe les commandes externes **pstree** et **top** qui sont aussi très utiles. **pstree** affiche les processus sous forme d'arbre généalogique. **top** affiche en temps réel la liste des processus actifs, réactualisée en permanence.

### Exemples

\$ **ps f**

PID	TTY	STAT	TIME	COMMAND
4722	pts/8	Ss	0:00	-bash
4767	pts/8	S	0:00	\_ aterm
4768	pts/10	Ss+	0:00	\_ bash
6002	pts/8	R+	0:00	\_ ps f

\$ **ps l**

F	UID	PID	PPID	PRI	NI	VSZ	RSS	WCHAN	STAT	TTY	TIME	COMMAND
0	5778	4722	4720	6	0	2192	1176	sigact	S	pts/8	0:00	-bash
0	5778	4767	4722	0	0	2404	1348	locks_	S	pts/8	0:00	aterm
0	5778	4768	4767	1	0	2148	1088	comple	S	pts/10	0:00	bash
0	5778	4791	4722	12	0	2524	872	-	R	pts/8	0:00	ps l

⇒ On obtient plus de renseignements, notamment le numéro d'utilisateur (UID) du propriétaire du processus, mais surtout le pid de son père (PPID). On remarque que **aterm** et **ps** ont le même père (le shell **bash**).

\$ **ps u**

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
cpb	4722	0.0	0.1	2192	3672	pts/8	Ss	09:43	0:00	-bash
cpb	4767	0.1	0.1	2404	3624	pts/8	S	11:17	0:00	aterm
cpb	4768	0.0	0.1	2148	3728	pts/10	Ss+	11:17	0:00	bash
cpb	6017	0.0	0.0	2524	1032	pts/8	R+	11:24	0:00	ps u

⇒ les mêmes processus avec un affichage orienté utilisateur.

\$ **ps lx**

F	UID	PID	PPID	PRI	NI	VSZ	RSS	WCHAN	STAT	TTY	TIME	COMMAND
4	5778	4696	4692	0	0	3940	2292	locks_	S	?	0:00	/usr/X11R6/
0	5778	4720	4696	0	0	2404	1356	-	R	?	0:00	aterm -ls
0	5778	4722	4720	7	0	2192	1176	sigact	S	pts/8	0:00	-bash
0	5778	4767	4722	0	0	2404	1348	locks_	S	pts/8	0:00	aterm
0	5778	4768	4767	1	0	2148	1088	comple	S	pts/10	0:00	bash
0	5778	6085	4722	13	0	2524	872	-	R	pts/8	0:00	ps lx

⇒ Les options d'affichage peuvent être combinées avec les options de sélection.

□

### 10.C.3 lsof : lister les fichiers ouverts

La commande externe **lsof** (*list open files*) permet de savoir quels sont les fichiers ouverts (utilisés) par les processus. Dans ce contexte, le terme de fichier ouvert est très large car il regroupe les fichiers réguliers/répertoire/spéciaux mais aussi les sockets internet et Unix, les bibliothèques partagées et les pipes, parmi d'autres choses encore. Il

faut dire que la philosophie Unix est que « *tout est fichier* »...

**ls**of admet un très grand nombre d'options permettant de sélectionner les processus/fichiers à afficher, ainsi que la manière de les afficher. Nous n'en étudions ici qu'une infime partie.

### Synopsis

```
lsof [-antFPU] {+D rép} [-f] {-u util} {-c prog} {-p pid} {-i [adresse]} [--] {nom}
```

 Si **ls**of n'est pas exécutée par root, de nombreuses informations/lignes seront manquantes...

Sans option ni argument, **ls**of affiche la liste de tous les *fichiers* utilisés par tous les processus, à raison d'un fichier et d'une utilisation par ligne. Les informations sont présentées en 9 colonnes (d'une largeur variable) :

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE/OFF	NODE	NAME
---------	-----	------	----	------	--------	----------	------	------

où :

- **COMMAND** : les 9 premiers caractères de la commande exécutée par le processus
- **PID** : le pid du processus
- **USER** : le nom (ou à défaut l'uid) de l'utilisateur possédant le processus
- **FD** : numéro du descripteur de fichier ou autre information, telle que `cwd` (répertoire de travail), `txt` (code d'un programme). **FD** peut être suivi d'un indicateur du mode d'ouverture : `r` pour lecture, `w` pour écriture, `u` pour les deux.
- **TYPE** : type associé au fichier. Parmi les possibilités, retenons `DIR` (répertoire), `REG` (fichier ordinaire), `CHR` (fichier spécial en mode caractère), `IPv4` (socket IPv4), `IPv6` (socket IPv6), ...
- **DEVICE** : pour des fichiers physiques, numéros indentifiant le disque. Pour d'autres types de fichiers, cela dépend ;
- **SIZE/OFF** : taille du fichier ou la position (*offset*) dans le fichier. L'*offset* est précédé de `0t` s'il est exprimé en décimal, et de `0x` s'il l'est en hexadécimal ;
- **NODE** : numéro d'*inode* pour un fichier, ou le protocole internet pour une socket ;
- **NAME** : le nom du fichier, ou l'adresse à laquelle est liée la socket.

### Exemple

Voici un très petit extrait (quelques lignes sur plus de 8000 alors que le système est très peu chargé) de la sortie de **ls**of, exécutée par root :

```
# ls
COMMAND  PID    USER   FD    TYPE    DEVICE  SIZE/OFF  NODE NAME
init      1      root   cwd    DIR      8,1     4096      2 /
init      1      root   rtd    DIR      8,1     4096      2 /
init      1      root   txt    REG      8,1     31676    741125 /sbin/init
init      1      root   10u    FIFO     0,5      0t0      2399 /dev/initctl
udevd     455    root    0u     CHR      1,3      0t0      550 /dev/null
udevd     455    root    8u     unix 0xf6fc0e00 0t0      2435 socket
portmap   1097   daemon  4u     IPv4     4539     0t0      UDP *:sunrpc
portmap   1097   daemon  5u     IPv4     4548     0t0      TCP *:sunrpc (LISTEN)
sshd      2026   root    3u     IPv4     6296     0t0      TCP *:ssh (LISTEN)
sshd      2026   root    4u     IPv6     6298     0t0      TCP *:ssh (LISTEN)
gnome-ter 2470   cyril   cwd    DIR      8,6     4096    10682369 /home/cyril
gnome-ter 2470   cyril   rtd    DIR      8,1     4096      2 /
gnome-ter 2470   cyril   txt    REG      8,1    260560    181918 /usr/bin/gnome-terminal
bash      8827   cyril   txt    REG      8,1    811156    993634 /bin/bash
bash      8827   cyril   mem    REG      8,1    42572    358579 /lib/i686/cmov/libnss_files-2.11.2.so
bash      8827   cyril   0r     CHR     136,7     0t0      10 /dev/pts/7
...
```

□

## Critères de sélection

**ls** permet d'exprimer des critères de sélection afin de réduire le nombre de fichiers affichés. Ces critères sont très nombreux et sont pour la plupart combinables (voir plus loin).

## Sélection selon le nom ou l'emplacement du fichier

En spécifiant un ou plusieurs *nom* en arguments, **ls** n'affichera que les processus utilisant l'un des fichiers mentionnés, d'une manière ou d'une autre. Cependant, si un *nom* correspond à un point de montage ou à un périphérique de type bloc (partition/disque) monté, la liste de tous les fichiers utilisés du système de fichiers correspondant sont affichés. L'option **-f** désactive ce comportement et ne retient que les processus utilisant directement ce *nom*.

✍ En cas d'utilisation de l'option **-f**, faire précéder le premier nom en argument de **--**, car **-f** prend elle-même des arguments. Pour éviter toute ambiguïté potentielle, mieux vaut systématiquement faire commencer la liste d'arguments par **--**.

L'option **+D rép** demande la liste des fichiers ouverts appartenant à l'arborescence du répertoire *rép*. Elle peut figurer plusieurs fois pour sélectionner les fichiers de plusieurs arborescences.

## Exemples

```
# ls -l /bin/sh
lrwxrwxrwx 1 root root 4 20 déc. 2010 /bin/sh -> dash
```

⇒ le fichier */bin/sh* est un lien symbolique vers */bin/dash*

```
# lsof /bin/sh
COMMAND    PID  USER  FD   TYPE DEVICE SIZE/OFF  NODE NAME
icedove    2687 cyril txt    REG   8,1    84144 993618 /bin/dash
run-mozil  2703 cyril txt    REG   8,1    84144 993618 /bin/dash
sh         9546  root txt    REG   8,1    84144 993618 /bin/dash
```

⇒ affiche la liste des processus utilisant */bin/sh* (en fait */bin/dash*)

```
# lsof /bin/sh /lib/ld-2.11.2.so
COMMAND    PID  USER  FD   TYPE DEVICE SIZE/OFF  NODE NAME
init        1    root  mem    REG   8,1   113964 342850 /lib/ld-2.11.2.so
portmap    1097 daemon mem    REG   8,1   113964 342850 /lib/ld-2.11.2.so
rpc.statd  1109  statd mem    REG   8,1   113964 342850 /lib/ld-2.11.2.so
rsyslogd   1325  root  mem    REG   8,1   113964 342850 /lib/ld-2.11.2.so
dbus-daem  1384 messagebus mem    REG   8,1   113964 342850 /lib/ld-2.11.2.so
x-session  2376  cyril mem    REG   8,1   113964 342850 /lib/ld-2.11.2.so
icedove    2687  cyril txt    REG   8,1    84144 993618 /bin/dash
icedove    2687  cyril mem    REG   8,1   113964 342850 /lib/ld-2.11.2.so
sh         9546  root txt    REG   8,1    84144 993618 /bin/dash
sh         9546  root mem    REG   8,1   113964 342850 /lib/ld-2.11.2.so
su        12282  root mem    REG   8,1   113964 342850 /lib/ld-2.11.2.so
...
```

⇒ affiche la longue liste (ici, tronquée) des fichiers utilisant */bin/sh* ou la librairie partagée */lib/ld-2.11.2.so*

```
# lsof /
COMMAND  PID  USER  FD   TYPE DEVICE SIZE/OFF  NODE NAME
init      1  root  cwd    DIR   8,1    4096      2 /
```



```

init      1    root  rtd    DIR     8,1      4096      2 /
init      1    root  txt    REG     8,1     31676   741125 /sbin/init
init      1    root  mem    REG     8,1     9736   358548 /lib/i686/cmov/libdl-2.11.2.so
init      1    root  mem    REG     8,1    1323460  358776 /lib/i686/cmov/libc-2.11.2.so
init      1    root  mem    REG     8,1    104276  342065 /lib/libselinux.so.1
...

```

⇒ / étant le point de montage d'une partition, tous les fichiers utilisés de la partition sont affichés.

```

# lsof -f -- /
COMMAND      PID      USER    FD    TYPE  DEVICE  SIZE/OFF  NODE NAME
init          1        root    cwd    DIR     8,1      4096      2 /
init          1        root    rtd    DIR     8,1      4096      2 /
kthreadd      2        root    cwd    DIR     8,1      4096      2 /
kthreadd      2        root    rtd    DIR     8,1      4096      2 /
...

```

⇒ alors qu'avec l'option **-f**, seuls les processus (certes nombreux) utilisant directement / sont affichés.

```

# lsof +D /var
COMMAND      PID      USER    FD    TYPE  DEVICE  SIZE/OFF  NODE NAME
portmap     1097    daemon   3uW    REG     8,1        5  48878 /var/run/portmap.pid
portmap     1097    daemon    6u    REG     8,1       74  48879 /var/run/portmap_mapping
rpc.statd   1109    statd    cwd    DIR     8,1     4096  131321 /var/lib/nfs
rpc.statd   1109    statd    5w    REG     8,1        5  48880 /var/run/rpc.statd.pid
...

```

⇒ affiche la liste des fichiers utilisés contenus dans l'arborescence de /var

□

## Sélection selon l'utilisateur

L'option **-u** *util* sélectionne les fichiers utilisés par l'utilisateur *util*. Elle peut être utilisée plusieurs fois, auquel cas les fichiers utilisés par l'un des utilisateurs mentionnés sont sélectionnés. Notons qu'*util* peut être une liste d'utilisateurs séparés par une virgule, ce qui revient au même que d'utiliser plusieurs fois l'option **-u**.

### Exemple

```

# lsof -u toto
COMMAND      PID  USER  FD  TYPE  DEVICE  SIZE/OFF  NODE NAME
gnome-key   2351  toto   cwd   DIR     8,1      4096   205625 /var/lib/gdm3
gnome-key   2351  toto   rtd   DIR     8,1      4096      2 /
gnome-key   2351  toto   txt   REG     8,1    754084  1166936 /usr/bin/gnome-keyring-daemon
gnome-key   2351  toto   mem   REG     8,1    48527   1166323 /usr/share/locale/fr/LC_MESSAGES/glib20.mo
...

```

⇒ affiche les fichiers utilisés par l'utilisateur toto

□

## Sélection selon le nom du programme

L'option **-c** *prog* demande d'afficher les fichiers utilisés par le processus exécutant un programme dont le nom commence par *prog*. Elle peut être utilisée plusieurs fois pour augmenter la sélection.

### Exemple

```
# lsof -c apache
```

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE/OFF	NODE	NAME
apache2	14697	root	cwd	DIR	8,1	4096	2	/
apache2	14697	root	rtd	DIR	8,1	4096	2	/
apache2	14697	root	txt	REG	8,1	398624	138699	/usr/lib/apache2/mpm-prefork/apache2
apache2	14697	root	mem	REG	8,1	71432	358539	/lib/i686/cmov/libresolv-2.11.2.so
apache2	14697	root	mem	REG	8,1	1392124	456148	/usr/lib/i686/cmov/libcrypto.so.0.9.8
apache2	14697	root	mem	REG	8,1	1216348	1165811	/usr/lib/libxml2.so.2.7.8

□

## Sélection selon le numéro de processus

L'option **-p** *pid* sélectionne les fichiers utilisés par le processus de numéro *pid*. Elle peut être utilisée plusieurs fois, pour augmenter la sélection. Notons que *pid* peut être une liste de numéros séparés par une virgule, ce qui revient au même que d'utiliser plusieurs fois l'option **-p**.

### Exemple

```
# lsof -p 20327
```

COMMAND	PID	USER	FD	TYPE	DEVICE	SIZE/OFF	NODE	NAME
bash	20327	cyril	cwd	DIR	8,6	4096	10682369	/home/cyril
bash	20327	cyril	rtd	DIR	8,1	4096	2	/
bash	20327	cyril	txt	REG	8,1	811156	993634	/bin/bash
bash	20327	cyril	mem	REG	8,1	42572	358579	/lib/i686/cmov/libnss_files-2.11.2.so
bash	20327	cyril	mem	REG	8,1	38504	358542	/lib/i686/cmov/libnss_nis-2.11.2.so
bash	20327	cyril	mem	REG	8,1	79676	358774	/lib/i686/cmov/libnsl-2.11.2.so
bash	20327	cyril	mem	REG	8,1	30496	358536	/lib/i686/cmov/libnss_compat-2.11.2.so
bash	20327	cyril	mem	REG	8,1	1527584	1181478	/usr/lib/locale/locale-archive
bash	20327	cyril	mem	REG	8,1	1323460	358776	/lib/i686/cmov/libc-2.11.2.so
bash	20327	cyril	mem	REG	8,1	9736	358548	/lib/i686/cmov/libdl-2.11.2.so
bash	20327	cyril	mem	REG	8,1	231576	342057	/lib/libncurses.so.5.7
bash	20327	cyril	mem	REG	8,1	26542	1164829	/usr/share/locale/fr/LC_MESSAGES/bash.mo
bash	20327	cyril	mem	REG	8,1	26048	368996	/usr/lib/gconv/gconv-modules.cache
bash	20327	cyril	mem	REG	8,1	113964	342850	/lib/ld-2.11.2.so
bash	20327	cyril	0r	CHR	136,10	0t0	13	/dev/pts/10
bash	20327	cyril	1u	CHR	136,10	0t0	13	/dev/pts/10
bash	20327	cyril	2u	CHR	136,10	0t0	13	/dev/pts/10
bash	20327	cyril	255u	CHR	136,10	0t0	13	/dev/pts/10

□

## Sélection des sockets Internet

L'option **-i** [*adresse*] sélectionne les processus qui utilisent une socket Internet liée à *adresse*. Si aucune adresse n'est mentionnée, cette option sélectionne toutes les sockets Internet. L'option **-i** peut être utilisée plusieurs fois pour augmenter le nombre de sockets sélectionnées.

L'argument *adresse* est de la forme :

[ **4** | **6** ] [ *protocole* ] [ @*hôte* ] [ : *service* ]

où toutes les parties sont optionnelles et :

- **4** ou **6** : précise la version d'IP ;
- *protocole* : précise le protocole de transport (comme UDP ou TCP) ;
- *hôte* : précise le nom ou l'adresse de l'hôte. Une adresse IPv4 s'écrit en notation décimale pointée, alors qu'une adresse IPv6 s'écrit entre crochets avec des : (ex : [3ffe:1ebc::1]). Dans le cas d'un protocole orienté connexion, l'*hôte* mentionné peut être distant ;
- *service* : précise le nom du service (tel que défini dans `/etc/services`) ou le numéro de son port. Une liste de services ou d'intervalles séparés par des virgules est admise.

L'option **-n** demande de ne pas réaliser la résolution de noms (transformation des adresses en noms d'hôte). L'option **-P** demande de ne pas réaliser la résolution des noms de services (transformation des numéros de ports en noms de services).

## Exemples

```
# lsof -i
COMMAND  PID      USER    FD  TYPE DEVICE SIZE/OFF NODE NAME
portmap   1058    daemon   4u  IPv4  4408      0t0  UDP *:sunrpc
portmap   1058    daemon   5u  IPv4  4417      0t0  TCP *:sunrpc (LISTEN)
rpc.statd 1070     statd    4w  IPv4  4442      0t0  UDP *:822
rpc.statd 1070     statd    6u  IPv4  4451      0t0  UDP *:52318
rpc.statd 1070     statd    7u  IPv4  4454      0t0  TCP *:45005 (LISTEN)
apache2   1433     root     3u  IPv6  5304      0t0  TCP *:www (LISTEN)
apache2   1460    www-data 3u  IPv6  5304      0t0  TCP *:www (LISTEN)
apache2   1461    www-data 3u  IPv6  5304      0t0  TCP *:www (LISTEN)
cupsd     1606     root     5u  IPv6  5755      0t0  TCP localhost:ipp (LISTEN)
cupsd     1606     root     6u  IPv4  5756      0t0  TCP localhost:ipp (LISTEN)
cupsd     1606     root     8u  IPv4  5759      0t0  UDP *:ipp
exim4     1899    Debian-exim 3u  IPv4  5952      0t0  TCP localhost:smtp (LISTEN)
exim4     1899    Debian-exim 4u  IPv6  5953      0t0  TCP localhost:smtp (LISTEN)
inetd     1976     root     4u  IPv4  6111      0t0  UDP *:tftp
hddtemp   2037     root     0u  IPv4  6305      0t0  TCP localhost:7634 (LISTEN)
sshd      2118     root     3u  IPv4  6431      0t0  TCP *:ssh (LISTEN)
sshd      2118     root     4u  IPv6  6433      0t0  TCP *:ssh (LISTEN)
firefox-b 2745     cyril    48u  IPv4  25716     0t0  TCP 172.16.203.109:52665->infodoc.iut.univ-aix.fr:www (ESTABLIS
firefox-b 2745     cyril    52u  IPv4  25719     0t0  TCP 172.16.203.109:52666->infodoc.iut.univ-aix.fr:www (ESTABLIS
dhclient  5317     root     6u  IPv4  23712     0t0  UDP *:bootpc
ssh       5669     cyril    3u  IPv4  25542     0t0  TCP 172.16.203.109:60270->allegro.iut.univ-aix.fr:ssh (ESTABLIS
ssh       5669     cyril    4u  IPv6  25547     0t0  TCP localhost:50000 (LISTEN)
ssh       5669     cyril    5u  IPv4  25548     0t0  TCP localhost:50000 (LISTEN)
...
```

⇒ Affiche les informations sur toutes les sockets Internet ouvertes. Certaines utilisent IPv4, d'autres IPv6. On peut remarquer que les adresses et les ports sont résolus lorsque possible, c'est à dire remplacés par des noms d'hôte et de services. Cela peut ralentir l'écriture du résultat.

```
# lsof -n -P -i          (ou lsof -nP -i)
COMMAND  PID  USER  FD  TYPE DEVICE SIZE/OFF NODE NAME
portmap   1058 daemon  4u  IPv4  4408      0t0  UDP *:111
...
apache2   1433  root   3u  IPv6  5304      0t0  TCP *:80 (LISTEN)
...
firefox-b 2745  cyril  48u  IPv4  25716     0t0  TCP 172.16.203.109:52665->139.124.187.14:80 (ESTABLISHED)
...
```

⇒ Accélère sensiblement le traitement en désactivant la résolution des noms d'hôte (**-n**) et celle des services (**-P**).

```
# lsof -n -i @139.124.187.4
COMMAND  PID  USER  FD  TYPE DEVICE SIZE/OFF NODE NAME
ssh       5669 cyril   3u  IPv4  25542     0t0  TCP 172.16.203.109:60270->139.124.187.4:ssh (ESTABLISHED)
```

⇒ Affiche les connexions à l'adresse (IPv4) 139.124.187.4.

```
# lsof -n -P -i @allegro.iut.univ-aix.fr
COMMAND  PID  USER  FD  TYPE DEVICE SIZE/OFF NODE NAME
ssh       5669 cyril   3u  IPv4  25542     0t0  TCP 172.16.203.109:60270->139.124.187.4:22 (ESTABLISHED)
```

⇒ La même commande mais en utilisant le nom de l'hôte plutôt que son adresse.

```
# lsof -n -i :25
COMMAND  PID      USER    FD  TYPE DEVICE SIZE/OFF NODE NAME
exim4    1899    Debian-exim 3u  IPv4  5952      0t0  TCP 127.0.0.1:smtp (LISTEN)
exim4    1899    Debian-exim 4u  IPv6  5953      0t0  TCP [::1]:smtp (LISTEN)
```

⇒ Affiche les sockets utilisant le port 25.

```
# lsof -n -i :1-1024
COMMAND  PID      USER  FD  TYPE DEVICE SIZE/OFF NODE NAME
portmap  1058    daemon  4u  IPv4  4408      0t0  UDP *:sunrpc
portmap  1058    daemon  5u  IPv4  4417      0t0  TCP *:sunrpc (LISTEN)
rpc.statd 1070    statd   4w  IPv4  4442      0t0  UDP *:822
apache2  1433    root    3u  IPv6  5304      0t0  TCP *:www (LISTEN)
apache2  1460    www-data 3u  IPv6  5304      0t0  TCP *:www (LISTEN)
apache2  1461    www-data 3u  IPv6  5304      0t0  TCP *:www (LISTEN)
cupsd    1606    root    5u  IPv6  5755      0t0  TCP [::]:ipp (LISTEN)
cupsd    1606    root    6u  IPv4  5756      0t0  TCP 127.0.0.1:ipp (LISTEN)
cupsd    1606    root    8u  IPv4  5759      0t0  UDP *:ipp
exim4    1899    Debian-exim 3u  IPv4  5952      0t0  TCP 127.0.0.1:smtp (LISTEN)
exim4    1899    Debian-exim 4u  IPv6  5953      0t0  TCP [::]:smtp (LISTEN)
inetd    1976    root    4u  IPv4  6111      0t0  UDP *:tftp
sshd     2118    root    3u  IPv4  6431      0t0  TCP *:ssh (LISTEN)
sshd     2118    root    4u  IPv6  6433      0t0  TCP *:ssh (LISTEN)
dhclient 5317    root    6u  IPv4  23712     0t0  UDP *:bootpc
ssh      5669    cyril   3u  IPv4  25542     0t0  TCP 172.16.203.109:60270->139.124.187.4:ssh (ESTABLISHED)
```

⇒ Affiche les sockets utilisant les ports compris entre 1 et 1024

```
# lsof -n -i 6
COMMAND  PID      USER  FD  TYPE DEVICE SIZE/OFF NODE NAME
apache2  1433    root    3u  IPv6  5304      0t0  TCP *:www (LISTEN)
apache2  1460    www-data 3u  IPv6  5304      0t0  TCP *:www (LISTEN)
cupsd    1606    root    5u  IPv6  5755      0t0  TCP [::]:ipp (LISTEN)
exim4    1899    Debian-exim 4u  IPv6  5953      0t0  TCP [::]:smtp (LISTEN)
sshd     2118    root    4u  IPv6  6433      0t0  TCP *:ssh (LISTEN)
ssh      5669    cyril   4u  IPv6  25547     0t0  TCP [::]:50000 (LISTEN)
```

⇒ affiche les sockets utilisant IPv6

```
# lsof -n -P -i @127.0.0.1
COMMAND  PID      USER  FD  TYPE DEVICE SIZE/OFF NODE NAME
cupsd    1606    root    6u  IPv4  5756      0t0  TCP 127.0.0.1:631 (LISTEN)
exim4    1899    Debian-exim 3u  IPv4  5952      0t0  TCP 127.0.0.1:25 (LISTEN)
hddtemp  2037    root    0u  IPv4  6305      0t0  TCP 127.0.0.1:7634 (LISTEN)
ssh      5669    cyril   5u  IPv4  25548     0t0  TCP 127.0.0.1:50000 (LISTEN)
```

⇒ Affiche les sockets utilisant l'adresse de rebouclage, ce qui comprend les services locaux.

□

## Sélection des sockets Unix

L'option **-U** sélectionne les sockets Unix, qui sont un moyen rapide de communication interne entre processus (rôle des fichiers spéciaux de type socket) très utilisé sur un système Unix. Il y a facilement plusieurs centaines socket Unix utilisées sur un système normal.

### Exemple

```
# lsof -U
COMMAND  PID      USER  FD  TYPE  DEVICE SIZE/OFF NODE NAME
udev     400     root    4u  unix 0xf6fe5e00      0t0 2240 socket
udev     400     root    8u  unix 0xf6fe5800      0t0 2244 socket
udev     400     root    9u  unix 0xf6fe5600      0t0 2245 socket
rsyslogd 1294    root    0u  unix 0xf6184200      0t0 4799 /dev/log
dbus-daem 1345    messagebus 3u  unix 0xf61ee200      0t0 4933 /var/run/dbus/system_bus_socket
dbus-daem 1345    messagebus 6u  unix 0xf61ef200      0t0 4938 socket
dbus-daem 1345    messagebus 7u  unix 0xf61ef400      0t0 4939 socket
dbus-daem 1345    messagebus 8u  unix 0xf6185e00      0t0 4960 /var/run/dbus/system_bus_socket
...
```

⇒ Le nombre de socket Unix sur un système standard est bien trop grand pour être affiché ici. Les noms de fichiers qui apparaissent dans la colonne NAME sont des fichiers spéciaux de type socket, comme le montre la commande ci-dessous :

```
# ls -l /dev/log /var/run/dbus/system_bus_socket
srw-rw-rw- 1 root root 0 12 sept. 15:07 /dev/log
srwxrwxrwx 1 root root 0 12 sept. 15:07 /var/run/dbus/system_bus_socket
```

□

## Combiner les sélections

Les critères de sélection peuvent être combinés. Par défaut, **lsof** traite plusieurs critères comme leur disjonction (le "ou"). Par exemple, la commande :

```
lsof -u toto -i
```

sélectionne tous les fichiers utilisés par l'utilisateur toto **plus** toutes les sockets internet (de tout utilisateur).

L'option **-a** demande de traiter les critères de natures différentes comme une conjonction (le "et"). Ainsi, la commande :

```
lsof -a -u toto -i
```

sélectionne uniquement les sockets internet utilisées par toto.

Cependant, même avec l'option **-a**, les critères de même nature sont traités comme une disjonction. Par exemple, en complétant la commande précédente ainsi :

```
lsof -a -u toto -i -u titi
```

on sélectionne les sockets internet utilisées par toto ou par titi.

## Contrôler la sortie de lsof

Outre les options **-n** et **-P** vues précédemment, **lsof** fournit plusieurs options permettent de contrôler la manière dont les informations sont écrites. En particulier, l'option **-F** permet de définir un format de sortie qui puisse être facilement récupérable par un autre programme (notamment via un pipe). Son utilisation peut être assez complexe et n'est pas décrite ici. Se reporter à la documentation.

Dans cet esprit, l'option **-t** demande de n'afficher que les numéros de processus correspondant, sans ligne d'en-tête. Cette option est particulièrement utile lorsqu'on souhaite envoyer un signal à ces processus.

## Exemple

```
# lsof -i @139.124.187.4 -n
COMMAND  PID  USER  FD  TYPE DEVICE SIZE/OFF NODE NAME
ssh      5669 cyril   3u  IPv4  25542      0t0  TCP 172.16.203.109:60270->139.124.187.4:ssh (ESTABLISHED)
ssh      6780 cyril   3u  IPv4  45085      0t0  TCP 172.16.203.109:33855->139.124.187.4:ssh (ESTABLISHED)
```

```
# lsof -i @139.124.187.4 -t
5669
6780
```

⇒ L'option **-t** ne fait afficher que les numéros des processus ayant une connexion internet avec l'adresse 139.124.187.4

```
# kill -TERM $(lsof -i @139.124.187.4 -t)
```

⇒ Envoie le signal **SIGTERM** à ces processus, obtenus par substitution de la commande **lsof** (voir section 18.B).

□

## 10.D Gestion des processus

### 10.D.1 Agir sur un processus avec les caractères de contrôle

Lorsqu'un processus s'exécute en premier plan, certains caractères dits de contrôle permettent d'agir sur son déroulement. Trois sont indiqués ci-dessous (il y en a d'autres) :

`CTRL-C` provoque la mort du processus (si celui-ci le veut bien) ;


`CTRL-\` provoque la mort du processus (même s'il ne le veut pas) ;

`CTRL-Z` provoque l'interruption du processus, dont on pourra déclencher ensuite la continuation en exécutant la commande interne **fg** pour continuer en premier-plan, ou **bg** pour continuer en tâche de fond.

### 10.D.2 kill : envoyer un signal à un processus

Les signaux font partie des communications inter-processus (IPC). Un signal est une information élémentaire envoyée à un processus par un autre processus via le système. Chaque signal est un numéro qui en donne la signification. Il y a plus de trente signaux "standards" sur Linux. La réception d'un signal par un processus peut conduire à son arrêt, sa suspension ou sa continuation (s'il était suspendu ou si le signal est ignoré).

Pour chaque signal, il y a un comportement par défaut, prévu par le système. Un processus peut le modifier, et ignorer un signal qui l'aurait fait se terminer. Il peut aussi associer à un signal une fonction à exécuter. À la réception de ce signal, il sera interrompu pour exécuter cette fonction avant de reprendre son exécution là où elle en était. Certains signaux ne peuvent pas être ignorés ni "déroutés", et conduisent forcément à l'arrêt du processus (**SIGKILL**) ou sa suspension (**SIGSTOP**). Lors de sa création, un processus hérite de la gestion des signaux de son père.

 Un processus peut envoyer un signal à n'importe quel processus du même utilisateur. Un processus root peut envoyer un signal à tous les processus du système.

Pour un utilisateur ou un administrateur système, les signaux sont pratiques pour agir sur des processus qui s'exécutent en tâches de fond (ou même en avant-plan) : pour les arrêter, les suspendre, ou leur demander de prendre en compte un changement de configuration. La commande interne <sup>4</sup> **kill** sert à envoyer des signaux à des processus.

#### Synopsis


```
kill -l [signal]
kill [-n] (pid | %job) {pid | %job}
```

Sans argument, l'option **-l** affiche la liste des signaux disponibles. Si elle est suivie d'un *signal*, il est traduit de nom en numéro ou inversement.

*pid* est le numéro d'un processus à qui on adresse le signal numéro *n* (*n* peut être aussi un nom de signal, voir ci-après). À la place du numéro de processus, on peut indiquer le numéro de job en utilisant *%job*. Mais dans ce cas, le job doit être un fils du shell où est tapée la commande **kill**.

Seuls quelques signaux parmi les nombreux disponibles sont particulièrement utiles et présentés ici.

4. Il existe aussi une commande externe **kill**.

 Taper `man 7 signal` pour avoir une description des signaux disponibles.

## Signal destiné à recharger la configuration

**1** ou **[SIG] HUP** les démons<sup>5</sup> recevant ce signal relisent (généralement) leur fichier de configuration. Un administrateur l'utilise ainsi chaque fois qu'il a modifié la configuration d'un service afin qu'elle soit prise en compte par le processus.

Néanmoins, **SIGHUP** est aussi le signal envoyé lorsqu'il y a une déconnexion d'un terminal et le comportement par défaut d'un processus le recevant est de s'arrêter. Lorsqu'on se déconnecte d'un terminal, le shell reçoit ce signal et le répercute à tous ses jobs encore en activité. S'ils ne se sont pas immunisés, cela provoque leur arrêt. C'est une façon de faire le ménage.

Pour éviter qu'un processus ne soit arrêté à la déconnexion du terminal, il faut avoir lancé sa commande en la précédant de **nohup** :

**nohup** *commande*


**nohup** est une commande externe qui exécute la *commande* en argument en immunisant son processus du signal **SIGHUP**.

## Signaux destinés à arrêter un processus

**15** ou **[SIG] TERM** signal gentil, demandant aux processus atteints de se terminer proprement. C'est le signal par défaut et *a priori* celui qu'il faut préférer pour arrêter un processus ;

**2** ou **[SIG] INT** signal plus brutal demandant un arrêt immédiat. C'est celui généralement provoqué par `CTRL-C`. À utiliser si le précédent n'a pas marché ;

**9** ou **[SIG] KILL** À utiliser en dernier recours car c'est le signal le plus violent. Il ne peut être intercepté par les processus et provoque la mort sans condition des processus atteints. A le même effet que `CTRL-\`.

 Si `-n` n'est pas spécifié (ni l'option `-l`), le signal pris par défaut est **SIGTERM**.

## Signaux destinés à suspendre/reprendre l'exécution d'un processus


**19** ou **[SIG] STOP** ne peut être ignoré et a le même effet que `CTRL-Z` ;

**20** ou **[SIG] TSTP** est le signal reçu par `CTRL-Z` ;

**18** ou **[SIG] CONT** reprendre l'exécution si le processus était suspendu, ignoré sinon.

## Exemples

\$ **bash**

 *lancement d'un bash en séquentiel.*

5. (en anglais, *daemons*) Programmes tournant en tâche de fond et rendant des services.



```
$ ps
  PID TTY          TIME CMD
 2953 pts/2    00:00:00 bash
 3130 pts/2    00:00:00 bash
 3152 pts/2    00:00:00 ps
```

⇒ On peut se douter que c'est celui de PID 3130 car ces numéros sont affectés par ordre croissant.

```
$ ps 1
  F   UID     PID   PPID  PRI  NI   VSZ   RSS  WCHAN  STAT TTY          TIME COMMAND
000   5778    2953    2918   11   0   2192  1176 sigact S     pts/2    0:00 bash
000   5778    3130    2953   13   0   2404  1348 sigact S     pts/2    0:00 bash
000   5778    3153    3130   18   0   2524   872  -      R     pts/2    0:00 ps 1
```

⇒ On le vérifie en consultant les PID/PPID : le PPID du processus **ps** est bien 3130

```
$ kill -l
 1) SIGHUP          2) SIGINT          3) SIGQUIT         4) SIGILL
 5) SIGTRAP        6) SIGABRT        7) SIGBUS          8) SIGFPE
 9) SIGKILL       10) SIGUSR1       11) SIGSEGV        12) SIGUSR2
13) SIGPIPE      14) SIGALRM      15) SIGTERM       16) SIGSTKFLT
17) SIGCHLD     18) SIGCONT     19) SIGSTOP      20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG       24) SIGXCPU
25) SIGXFSZ     26) SIGVTALRM   27) SIGPROF      28) SIGWINCH
29) SIGIO       30) SIGPWR      31) SIGSYS        34) SIGRTMIN
35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3    38) SIGRTMIN+4
39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7    42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11   46) SIGRTMIN+12
47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15   50) SIGRTMAX-14
51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11   54) SIGRTMAX-10
55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7    58) SIGRTMAX-6
59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3    62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

⇒ Consultation des signaux disponibles (ou enlèvera **SIG** pour les utiliser).

```
$ kill 3130
$ ps
  PID TTY          TIME CMD
 2953 pts/2    00:00:00 bash
 3130 pts/2    00:00:00 bash
 3154 pts/2    00:00:00 ps
```

⇒ le signal **TERM** n'a pas été suffisant pour tuer ce bash.

```
$ kill -INT 3130
$ ps
  PID TTY          TIME CMD
 2953 pts/2    00:00:00 bash
 3130 pts/2    00:00:00 bash
 3155 pts/2    00:00:00 ps
```

⇒ le signal **INT** ne l'a pas été non plus.

```
$ kill -9 3130
Killed
$ ps
  PID TTY          TIME CMD
```



```
2953 pts/2      00:00:00 bash
3156 pts/2      00:00:00 ps
```

⇒ en revanche, **KILL** l'a été.

□

### 10.D.3 pkill : envoyer un signal à des processus sélectionnés

La commande **pkill** permet d'envoyer un signal aux processus répondant à certains critères.

#### Synopsis

```
pkill [-signal] [-fv (n|o)] [-u eid{ , eid}] [-g egid{ , egid}] [-U ruid{ , ruid}]
      [-G rgid{ , rgid}] [-P ppid{ , ppid}] [-t term{ , term}] motif
```

**pkill** recherche dans la liste des processus, ceux qui vérifient tous les critères mentionnés et leur envoie le signal *signal* (par défaut **SIGTERM**).

Au moins l'un des critères suivants doit être mentionné :

- *motif* est une expression régulière étendue. Un processus satisfait ce critère si le nom du programme qu'il exécute correspond à ce *motif*. Si l'option **-f** est spécifiée, la comparaison se fait sur l'ensemble de la ligne de commande exécutée par le processus ;
- **-u** *eid*{ , *eid*} : un processus satisfait ce critère si son propriétaire (UID) effectif est l'un des *eid* mentionnés, où *eid* est un nom ou un numéro d'utilisateur ;
- **-g** *egid*{ , *egid*} : un processus satisfait ce critère si son groupe effectif est l'un des *egid* mentionnés, où *egid* est un nom ou un numéro de groupe. La valeur 0 pour *egid* indique le même groupe que celui du processus exécutant **pkill** ;
- **-U** *ruid*{ , *ruid*} : un processus satisfait ce critère si son propriétaire (UID) réel est l'un des *ruid* mentionnés ;
- **-G** *rgid*{ , *rgid*} : un processus satisfait ce critère si son groupe réel est l'un des *rgid* mentionnés ;
- **-P** *ppid*{ , *ppid*} : un processus satisfait ce critère si le PID de son père est l'un des *ppid* mentionnés ;
- **-t** *term*{ , *term*} : un processus satisfait ce critère s'il est rattaché à l'un des terminaux *term* (spécifiés sans le préfixe /dev/).

L'option **-v** demande d'inverser la sélection. L'option **-n** demande de ne retenir que le processus le plus récent satisfaisant les critères, alors que l'option **-o** demande de ne retenir que le plus ancien.

### 10.D.4 killall : envoyer un signal aux processus exécutant certaines commandes

La commande **killall** permet d'envoyer un signal à des processus exécutant certaines commandes.

#### Synopsis

```
killall [-liqrv] [-s signal] [-u utilisateur] [--] nom {nom}
```

**killall** recherche les processus exécutant une commande correspondant à l'un des *noms* spécifiés et leur envoie le *signal* indiqué par l'option **-s** (par défaut **SIGTERM**). L'option **-l** (*i* majuscule) demande à ce que la recherche se fasse sans tenir compte de la casse. L'option **-r** demande à traiter les *noms* comme des expressions régulières étendues. L'option **-u** demande de n'envoyer un signal qu'aux processus de l'utilisateur *utilisateur*. Dans ce cas, la liste de *noms* est optionnelle. L'option **-i** demande une confirmation avant d'envoyer un signal à chacun des processus trouvés. L'option **-v** demande d'afficher un message pour chaque envoi de signal réussi. L'option **-q** demande de ne pas signaler les envois qui ont échoué.



# Chapitre 11

## Utilisateurs et groupes

---

### 11.A Gestion des comptes utilisateurs

#### 11.A.1 Les fichiers `/etc/passwd` et `/etc/shadow`

Le fichier recensant les utilisateurs est le fichier texte `/etc/passwd`. Il contient une ligne par utilisateur existant. Chaque ligne comporte sept champs séparés par un `:` (*deux-points*) et a le format suivant :

*nom : motdepasse : uid : gid : infos : répertoire : shell*

où :

*nom* le nom de l'utilisateur, composé d'au maximum 8 caractères alphanumériques. Ce nom doit être unique ;

*motdepasse* le mot de passe crypté de l'utilisateur. En principe, par mesure de sécurité, ce champ ne contient que le caractère **x**, indiquant que le mot de passe crypté de cet utilisateur est contenu dans le fichier `/etc/shadow` qui n'est lisible que par root.


*uid* le numéro de l'utilisateur qui doit être unique. Ce numéro va de pair avec le nom d'utilisateur ;

*gid* le numéro de groupe d'appartenance par défaut (groupe principal) de cet utilisateur. Ce groupe doit être défini dans le fichier `/etc/group` (voir ci-après). Notons qu'un utilisateur peut appartenir à plusieurs groupes. Pour ses groupes secondaires, il doit figurer comme membre dans le fichier `/etc/group` (voir plus loin) ;

*infos* ce champ, appelé aussi *gecos*, contient en principe le nom réel de l'utilisateur auquel on peut ajouter le numéro du bureau, le numéro de téléphone au bureau et celui à la maison. Ces informations sont séparées par des virgules. Tout utilisateur peut modifier les informations le concernant en utilisant la commande **chfn** (*change finger*). Historiquement ces informations étaient disponibles à travers le réseau en utilisant la commande **finger** mais en raison des problèmes de confidentialité que cela pose, cette possibilité n'est presque plus offerte ;

*répertoire* le répertoire personnel (ou d'accueil) de l'utilisateur ;

*shell* le shell par défaut utilisé par l'utilisateur. Il s'agit d'un chemin absolu vers l'interpréteur de commande. Les différents shells disponibles sont en principe recensés dans le fichier `/etc/shells`. Tout utilisateur peut spécifier le shell qu'il souhaite utiliser par défaut en utilisant la commande **chsh** (*change shell*).

 Puisque le `:` est un séparateur de champ dans les fichiers `/etc/passwd` et `/etc/shadow`, il ne peut figurer dans aucun champ.

Il suffit que root ajoute une ligne au fichier `/etc/passwd` (et au fichier `/etc/shadow`) pour créer un utilisateur.

## 11.A.2 useradd : créer un utilisateur


La commande **useradd** facilite la tâche de création d'un utilisateur en modifiant les fichiers `/etc/passwd` et `/etc/shadow`. Elle n'est utilisable que par root.

### Synopsis


```
useradd [-u uid] [-r] [-g groupe] [-c infos] [-d rép] [-s shell] [-G groupes] nom
```


Les options permettent de renseigner les différents champs du fichier `/etc/passwd`. L'option **-r** demande la création d'un utilisateur "système" : utilisateur fictif qui ne dispose pas normalement de répertoire personnel. Dans ce cas, le système choisira pour numéro d'utilisateur, le premier numéro libre inférieur à 500. L'option **-G** permet de spécifier d'autres groupes d'appartenance pour l'utilisateur. Elle a pour conséquence de modifier le fichier `/etc/group` (voir section 11.B.1 page 136).

Pour chaque champ, il existe des valeurs par défaut contenues dans le fichier `/etc/default/useradd`. Le comportement de **useradd** elle-même est défini par le fichier `/etc/login.defs`. Notamment, ce fichier précise l'emplacement de la boîte aux lettres des utilisateurs, s'il faut créer un répertoire pour chaque nouvel utilisateur (normalement, oui), etc.

 On remarquera qu'il n'est pas indiqué d'option permettant de préciser le mot de passe de l'utilisateur. Il y en a bien une mais son utilisation est complexe. C'est pourquoi une création d'utilisateur est généralement suivie de la modification de son mot de passe en tapant **passwd nom**.

Enfin, lorsque le répertoire d'un nouvel utilisateur est créé comme une copie du répertoire `/etc/skel`. Les fichiers qu'il contient peuvent être de nature quelconque mais correspondent généralement à des fichiers de configuration.

 Il existe les commandes **usermod** et **userdel** permettant respectivement de modifier et de supprimer un utilisateur.

 Notons qu'il existe une version spéciale Debian (et dérivés) pour la création d'utilisateurs. Son nom est **adduser**. Elle s'utilise de manière sensiblement différente.

## 11.A.3 Modification du mot de passe utilisateur

### 11.A.3.a passwd : changer son mot de passe local

Sous Unix, la commande externe permettant de modifier le mot de passe d'un utilisateur est **passwd**.

### Synopsis

```
passwd [-edl] [-S [-a]] [utilisateur]
```

Sans option, ni argument, **passwd** modifie le mot de passe de l'utilisateur en session. Pour cela, il lui demande son ancien mot de passe puis deux fois le nouveau. Si tout concorde et que le nouveau mot de passe est conforme à la politique de sécurité, il est mis à jour dans `/etc/shadow`. C'est à peu près tout ce que peut faire un utilisateur sans privilège.

root peut changer le mot de passe de l'*utilisateur* en argument sans fournir son ancien mot de passe. Il peut aussi le faire expirer avec l'option **-e**, obligeant l'utilisateur à le changer lors de sa prochaine connexion, ou empêcher l'utilisation de son mot de passe avec l'option **-d**, voire carrément bloquer son utilisation et sa modification par l'utilisateur avec l'option **-l**.

L'option **-S** affiche l'état du mot de passe de l'*utilisateur*. Combinée à l'option **-a**, elle affiche l'état des mots de passe de tous les utilisateurs. Dans ce cas, il ne faut pas indiquer d'argument.

### Exemple

\$ **passwd**

Changement du mot de passe pour toto.

Mot de passe UNIX (actuel) :

Entrez le nouveau mot de passe UNIX :

Retapez le nouveau mot de passe UNIX :

passwd : le mot de passe a été mis à jour avec succès

⇒ *les mots de passe n'apparaissent pas à l'écran lorsqu'on les saisit*

# **passwd bidule**

Entrez le nouveau mot de passe UNIX :

Retapez le nouveau mot de passe UNIX :

passwd : le mot de passe a été mis à jour avec succès

⇒ *root n'a pas besoin de fournir le mot de passe d'un utilisateur pour le modifier*

□

### 11.A.3.b smbpasswd : changer son mot de passe avec SAMBA

Parfois, les mots de passe des utilisateurs (ainsi que leurs profils) sont centralisés et gérés par un serveur. Dans le monde Windows, un tel serveur est appelé un **contrôleur de domaine** et les utilisateurs appartiennent à un des **domaines** qu'il gère. La suite de logiciels SAMBA sous Unix, permet de partager ces informations entre les postes Windows (y compris serveur) et les postes Linux. SAMBA permet même à un poste Linux d'être le contrôleur de domaines.

Dans ce genre d'environnement, pour changer son mot de passe sous Linux, on n'utilise plus **passwd** mais on doit le faire en utilisant la commande externe **smbpasswd** conçue pour SAMBA.

### Synopsis

**smbpasswd** [**-r** *contrôleur*]

Lorsqu'on exécute cette commande, celle-ci demande le mot de passe de l'utilisateur en session puis, s'il est correct, elle demande de taper deux fois le nouveau mot de passe. Ce sera ensuite votre mot de passe (Windows NT et Linux). L'option **-r** *contrôleur* permet de spécifier le contrôleur de domaine qui gère les mots de passe.

### Exemple

bob@B98X:~\$ **smbpasswd -r uncontrôleur**

Old SMB password:

New SMB password:

Retype new SMB password:

Password changed for user bob

⇒ *changement du mot de passe de l'utilisateur bob, géré par le contrôleur uncontrôleur, depuis un poste Linux. Les mots de passe ne sont pas affichés lorsqu'on les tape.*

□

## 11.B Gestion des groupes



Avant de créer un utilisateur, il faut d'abord que son groupe ait été créé.

De même que pour les utilisateurs, la gestion des groupes est grandement facilitée car ils sont recensés dans de simples fichiers texte.

### 11.B.1 Les fichiers `/etc/group` et `/etc/gshadow`

Le fichier recensant les groupes est le fichier texte `/etc/group`. Il contient une ligne par groupe créé. Chaque ligne comporte quatre champs séparés par un `:` et a le format suivant :

*nom* : *motdepasse* : *gid* : *membres*

*nom* le nom du groupe ;

*motdepasse* : le mot de passe crypté du groupe. Son utilité sera expliquée en même temps que la commande **newgrp** (section 11.D.2 page 141). Le plus souvent, par mesure de sécurité, les mots de passe des groupes sont contenus dans le fichier `/etc/gshadow`, qui n'est lisible que par root. Le champ *motdepasse* du fichier `/etc/group` ne contient alors que le caractère **x** ;

*gid* le numéro du groupe ;

*membres* parfois vide, ce champ contient la liste des utilisateurs qui appartiennent au groupe sans que celui-ci ne soit leur groupe d'appartenance par défaut. Les utilisateurs sont indiqués par leur nom ou leur numéro, et séparés par des virgules.

Il suffit que root ajoute une ligne au fichier `/etc/group` (et si besoin à `/etc/gshadow`) pour créer un groupe.



À nouveau, aucun champ ne peut contenir le caractère `:` (*deux-points*).

### 11.B.2 **groupadd** : créer un groupe

La commande externe **groupadd** facilite la tâche de création de groupe en modifiant les fichiers `/etc/group` et `/etc/gshadow`.

Elle n'est utilisable que par root.


#### **Synopsis**


**groupadd** [-g *gid*] [-r] *groupe*

L'argument *groupe* indique le nom du groupe à créer. Sans option, le système lui attribuera le premier numéro (*gid*) libre supérieur à 500. L'option **-r** demande la création d'un groupe "système" (représentant des utilisateurs fictifs, comme ceux qui utiliseront une imprimante, ou regroupant des serveurs). Dans ce cas, le système lui attribuera le premier numéro libre inférieur à 500. L'option **-g** *gid* permet de spécifier manuellement un numéro quelconque mais il ne doit pas déjà être utilisé.



Il n'existe pas d'option pour préciser le mot de passe de groupe lors de sa création. Il faudra utiliser la commande **gpasswd** vue section 11.D.1 page 141.

 Il existe les commandes **groupmod** permettant de modifier un groupe et **groupdel** permettant de supprimer un groupe. Bien entendu, elles ne sont utilisables que par root.

 Notons qu'il existe une version spéciale Debian (et dérivés) pour la création de groupes. Son nom est **addgroup**. Elle s'utilise de manière sensiblement différente.

## 11.C Gestion des identités

Dans le cas normal, un processus hérite des identités (propriétaire et groupes) de son père. Ainsi, lorsqu'un utilisateur tape une commande, le processus qui l'exécute est propriété de cet utilisateur et a ses groupes. Il est limité dans ses actions par les droits conférés à ces identités. Ce mécanisme participe à la sécurité d'un système Unix.

Néanmoins, il est parfois nécessaire d'**élever le niveau de privilège** d'un processus utilisateur : soit pour réaliser une tâche élémentaire qui ne peut être effectuée sans privilège, comme changer son mot de passe, soit réaliser des tâches d'administration sans ouvrir de session root.

Unix propose un mécanisme simple pour exécuter un processus sous une autre identité (avec ou sans privilège). Il repose sur les fichiers exécutables qui peuvent être configurés pour s'exécuter sous une autre identité dans certaines conditions. Cette méthode est utilisée pour les commandes que peuvent légitimement exécuter les utilisateurs mais qui requièrent un changement d'identité, telles que **passwd**, **gpasswd** ou **write**. Parmi ces commandes particulières, certaines comme **su** et **sudo** peuvent exécuter une autre commande (processus) en modifiant son identité.

### 11.C.1 Identités réelle et effective

Tout processus possède une identité réelle et une identité effective. L'identité réelle est celle ayant servi à exécuter un processus. Elle comprend l'utilisateur réel (son UID) et ses groupes réels. Elle est purement informative. L'identité effective est celle sous laquelle le processus est exécuté. C'est elle qui est utilisée par le système pour déterminer les droits du processus. Elle comprend l'utilisateur effectif (son UID) et ses groupes effectifs.

#### 11.C.1.a id : connaître son identité effective et réelle


Pour connaître son identité effective et réelle ou des informations sur l'identité d'un utilisateur, on peut utiliser la commande externe **id**.

##### Synopsis

```
id [-gnruG] [utilisateur]
```


Sans option ni argument, écrit l'**UID** effectif, le nom d'utilisateur associé, le **GID** effectif, le nom de groupe associé, ainsi que tous les groupes d'appartenance de l'utilisateur en session. Si *utilisateur* est spécifié, les informations par défaut pour cet utilisateur sont écrites.

Les options demandent une partie seulement de ces informations : **-g** pour le *gid*, **-u** pour l'*uid*, **-G** pour les groupes d'appartenance. L'option **-n** est à combiner avec les autres pour demander que le nom soit affiché et non le numéro (d'utilisateur ou de groupe). L'option **-r** demande l'identité réelle et non pas effective.

 Il est rare de trouver de nos jours une installation où les identités réelles et effectives sont différentes dans un shell. Pour des raisons de sécurité, des garde-fous ont été mis en place pour empêcher certaines élévations de privilèges.

## Exemples

```
$ id
uid=1000(toto) gid=100(users) groupes=100(users),7(lp),20(dialout),24(cdrom),
25(floppy),29(audio),44(video),46(plugdev),109(netdev),110(bluetooth),111(lpadmin),
113(fuse),116(scanner),118(powerdev)
$ id -u
1000
$ id -un
cyril
$ id -urn
cyril
$ id -G
100 7 20 24 25 29 44 46 109 110 111 113 116 118 1001
$ id -Gn
users lp dialout cdrom floppy audio video plugdev netdev bluetooth lpadmin fuse
scanner powerdev
$ id -gn
users
```

 bien que l'utilisateur possède plusieurs groupes, son groupe par défaut est users.



### 11.C.1.b whoami : connaître son nom d'utilisateur effectif

La commande **whoami** affiche le nom de l'utilisateur courant.


## Synopsis

**whoami**

Affiche le même résultat que la commande **id -un**.

## Exemple

```
$ whoami
marcel
$ id -un
marcel
```

 l'utilisateur en session est marcel.





### 11.C.1.c groups : connaître les groupes d'un utilisateur

Lorsqu'un utilisateur appartient à plusieurs groupes, il possède les permissions de tous ses groupes. Cela permet à plusieurs utilisateurs de partager des données, ou de travailler sur le même projet.

Par exemple, dans une entreprise, pour chaque projet en développement, on va créer un groupe et y inclure les utilisateurs travaillant sur ce projet. Les fichiers de ce projet seront alors de ce groupe et seront accessibles aux seuls membres du groupe.

Pour savoir à quels groupes un utilisateur appartient, il faut utiliser la commande externe **groups**.

#### Synopsis

**groups** { *utilisateur* }

Si aucun *utilisateur* n'est mentionné, indique les groupes d'appartenance de l'utilisateur en session. Sinon, pour chaque utilisateur mentionné, indique ses groupes.

#### Exemples

 Ces exemples ne sont probablement plus valables sur l'installation actuelle d'allegro.

```
cpb@allegro:~$ groups
cpb prof
```

➡ l'utilisateur en session (ici cpb) appartient aux groupes cpb et prof

```
cpb@allegro:~$ groups sucgre
sucgre etud1 de_jul
```

➡ et sucgre appartient aux groupes sucgre, etud1 et de\_jul

□

## 11.C.2 su : utiliser une autre identité pour exécuter un shell ou une commande

La commande **su** permet de se connecter ou d'exécuter une commande sous l'identité d'un autre utilisateur.

#### Synopsis

**su** [-[**l**]] [-**c** *commande*] [*utilisateur*]

Sans option ni argument, **su** ouvre un shell sous l'identité de root. Si un *utilisateur* est indiqué, ouvre un shell sous son identité. Le mot de passe de l'*utilisateur* ciblé est normalement demandé, sauf à root.

L'option - (ou -l) demande l'ouverture d'un login-shell afin d'avoir le même environnement qu'une connexion directe sous cette identité. L'option -c demande à ce qu'à la place d'un shell interactif, celui-ci exécute *commande* et se termine. Elle permet donc d'exécuter une commande sous l'identité de l'utilisateur ciblé. Notons que *commande* ne doit former qu'un seul mot, ce que l'on obtient si on l'encadre par des quotes ou des guillemets.

#### Exemples

```
$ whoami
```

```
marcel
```

```
$ pwd
```

```
/home/marcel
```

```
$ cat /etc/shadow
```

```
cat: /etc/shadow: Permission non accordée
```

⇒ *l'utilisateur marcel n'a pas le droit de visualiser /etc/shadow*

```
$ su -
```

```
Mot de passe :
```

```
# whoami
```

```
root
```

```
# pwd
```

```
/root
```

⇒ *En fournissant le mot de passe de root, marcel a ouvert un shell en tant que root. L'option - demande un login shell. Le répertoire de travail est alors /root.*

```
# cat /etc/shadow
```

```
root:hkjh8bBlmkjkj:14844:0:99999:7:::
```

```
daemon*:14844:0:99999:7:::
```

```
bin*:14844:0:99999:7:::
```

```
sys*:14844:0:99999:7:::
```

```
sync*:14844:0:99999:7:::
```

```
marcel:fsdggfdf:14844:0:99999:7:::
```

```
...
```

⇒ *root peut ouvrir /etc/shadow*

```
# su toto
```

```
$ whoami
```

```
toto
```

```
$ pwd
```

```
/root
```

⇒ *root peut ouvrir un shell sous n'importe quelle identité sans fournir de mot de passe. N'ayant pas demandé un login shell, le répertoire de travail reste ce qu'il était.*

```
$ exit
```

⇒ *fermeture du shell de toto pour revenir à celui de root*

```
# exit
```

```
logout
```

⇒ *fermeture du (login) shell root pour revenir à celui de marcel*

```
$ whoami
```

```
marcel
```

```
$ su -c 'cat /etc/shadow' root
```

```
Mot de passe :
```

```
root:hkjh8bBlmkjkj:14844:0:99999:7:::
```

```
daemon*:14844:0:99999:7:::
```

```
bin*:14844:0:99999:7:::
```

```
sys*:14844:0:99999:7:::
```

```
sync*:14844:0:99999:7:::
```

```
marcel:fsdggfdf:14844:0:99999:7:::
```

```
...
```

- ⇒ exécution de la commande **cat /etc/shadow** en tant que **root** sans passer par un shell interactif. Notons que la commande aurait pu être **vi /etc/shadow**, ou être une commande plus complexe.

□

✍ L'inconvénient majeur de **su** est qu'elle nécessite le mot de passe de l'utilisateur cible. Or, pour la sécurité du système, il n'est jamais bon que les utilisateurs aient connaissance d'autres mots de passe que le leur. C'est même généralement interdit par les chartes informatiques. Quant à la communication du mot de passe de **root**, cela se passe de commentaires. . .

L'alternative de plus en plus répandue est l'utilisation de la commande **sudo**. Elle permet aussi d'exécuter des commandes sous l'identité de **root** ou d'autres utilisateurs, mais le mot de passe à fournir (si besoin) est son propre mot de passe. En contrepartie, seules les commandes autorisées par **root** peuvent être exécutées par ce biais. Pour cela, **root** utilise la commande **visudo** pour éditer le fichier `/etc/sudoers`, et définit qui a le droit d'exécuter quelle(s) commande(s) sous quelle(s) identité(s).

## 11.D Travailler avec les groupes

### 11.D.1 gpasswd : administrer un groupe

Lors de la création d'un groupe, **root** peut désigner un ou plusieurs administrateurs pour ce groupe, qui auront alors la possibilité d'y ajouter et d'en supprimer des membres, et de modifier le mot de passe du groupe.

Toutes ces opérations se font en utilisant la commande externe **gpasswd**. Elle a plusieurs syntaxes d'utilisation.

#### Synopsis

**gpasswd [-A admin{ , admin}] [-M utilisateur{ , utilisateur}] groupe**

Utilisable par **root** **uniquement**, cette commande fixe les *admin* comme administrateurs du groupe *groupe* et les *utilisateur* comme membres ;

**gpasswd -a utilisateur groupe**

Utilisable aussi par un administrateur de *groupe* pour y ajouter le membre *utilisateur* ;

**gpasswd -d utilisateur groupe**

Utilisable aussi par un administrateur de *groupe* pour en supprimer le membre *utilisateur* ;

**gpasswd -R groupe**

Utilisable aussi par un administrateur de *groupe* pour en interdire l'accès aux non-membres par la commande **newgrp** (détaillée dans la section 11.D.2) ;

**gpasswd groupe**

Utilisable aussi par un administrateur de *groupe* pour en fixer un mot de passe. Les non-membres du groupe pourront y accéder avec la commande **newgrp** s'ils en connaissent le mot de passe.

### 11.D.2 newgrp : nouveau shell en changeant de groupe par défaut


Les utilisateurs ont chacun comme groupe principal celui qui est indiqué dans le fichier `/etc/passwd`. Ce groupe est utilisé par défaut, notamment lors de la création d'un fichier. On dira que c'est le **groupe effectif**.

### Exemple

Supposons qu'un utilisateur *toto* ait pour groupe principal *users*. S'il crée un fichier, ce fichier possèdera *users* comme groupe.

```
$ echo essai > fic
$ ls -l fic
-rw-r--r-- 1 toto  users 6 2007-08-28 10:12 fic
```




 Les autres groupes auxquels l'utilisateur appartient ne sont utilisés que pour déterminer les droits de celui-ci sur les fichiers/répertoires. S'il n'en est pas le propriétaire mais un membre de son groupe, il a les permissions du groupe.

La commande externe **newgrp** permet d'exécuter un shell dans lequel le groupe effectif est différent du groupe principal. Cela peut s'avérer pratique si l'on veut créer plusieurs fichiers avec un certain groupe, différent de son groupe principal.

### Synopsis

**newgrp** [*groupe*]

Sans argument, crée un shell où l'on prend comme groupe effectif, son groupe principal. Sinon, crée un shell avec pour groupe effectif, le *groupe* spécifié. Si l'utilisateur est membre de ce groupe, aucun mot de passe ne sera demandé. Sinon, le mot de passe du groupe sera demandé.

 Cette commande crée systématiquement un nouveau shell même si elle échoue.

## 11.E Informations sur les utilisateurs

### 11.E.1 who : savoir quels sont les utilisateurs connectés

La commande **who** affiche les informations sur les utilisateurs connectés mais aussi certaines informations sur le système.

### Synopsis

**who** [-bmqr] [--ips] [*mot*<sub>1</sub> *mot*<sub>2</sub>]

Sans option ni argument, **who** affiche la liste des utilisateurs connectés à un terminal, à raison d'une ligne par utilisateur et par terminal qu'il utilise. Chaque ligne contient, dans l'ordre et séparés par des blancs :

- le nom d'utilisateur ;
- le terminal auquel il est connecté, sans le préfixe */dev/* ;
- la date et l'heure de sa connexion ;
- un commentaire sur l'origine de la connexion : s'il s'agit d'une session graphique ou d'un affichage déporté, affiche le *display* qu'elle utilise. S'il s'agit d'une connexion de type terminal distant (SSH, TELNET, ...), affiche le nom (ou à défaut l'adresse) de l'hôte distant.

❗ **who** n’affiche que les utilisateurs possédant un terminal. Ceux qui utilisent uniquement des applications sans terminal n’apparaissent pas.

### Exemples

```
allegro$ who
leborgne pts/0      2011-09-12 08:51 (pcdl.iut.univ-aix.fr)
ambchr   pts/1      2011-09-15 08:18 (a75.iut.univ-aix.fr)
zekaur   pts/2      2011-09-15 08:19 (a74.iut.univ-aix.fr)
troseb   pts/3      2011-09-15 08:20 (a76.iut.univ-aix.fr)
jamsyl   pts/4      2011-09-15 08:37 (a82.iut.univ-aix.fr)
cpb      pts/5      2011-09-15 09:31 (ala02u11.u-3mrs.fr)
jamsyl   pts/6      2011-09-15 08:44 (a82.iut.univ-aix.fr)
chatom   pts/7      2011-09-15 08:44 (a80.iut.univ-aix.fr)
lublud   pts/8      2011-09-15 08:44 (a81.iut.univ-aix.fr)
```

⇒ On peut remarquer toutes les connexions affichées ici sont de type terminal distant, et qu’un utilisateur apparaît autant de fois que de terminaux qui lui sont attribués.

```
pc$ who
toto     tty7      2011-09-12 15:08 (:0)
toto     pts/1      2011-09-12 15:08 (:0.0)
bidule   2011-09-12 11:12 (139.124.187.124:0)
bidule   pts/2      2011-09-12 11:13 (139.124.187.124:0.0)
root     pts/4      2011-09-12 15:49 (:0.0)
cyril    pts/5      2011-09-12 15:50 (:0.0)
titi     pts/6      2011-09-15 10:06 (cluster-01)
```

⇒ Dans le cas de sessions graphiques ou d’affichage déporté, la partie commentaire affiche le display utilisé.

□

L’option **-m** n’affiche que la ligne correspondant au terminal sur laquelle **who** est exécutée. Elle renseigne donc sur l’utilisateur de ce terminal. Le même effet est obtenu en utilisant les arguments *mot*<sub>1</sub> et *mot*<sub>2</sub> (qui sont ignorés en présence d’option différente de **--ips**). L’habitude est qu’on utilise **am i** en arguments, mais tout couple de mots donne le même résultat.

### Exemples

```
$ who -m
toto     pts/1      2011-09-12 15:08 (:0.0)
$ who am i
toto     pts/1      2011-09-12 15:08 (:0.0)
$ who is who
toto     pts/1      2011-09-12 15:08 (:0.0)
```

⇒ Les 3 commandes sont équivalentes et affichent l’utilisateur possédant le terminal sur lequel elles sont tapées.

□

L’option **--ips** désactive la résolution de noms des adresses IP. L’option **-q** affiche la liste des utilisateurs sur une ligne en les séparant par des blancs, puis une ligne indiquant leur nombre.

### Exemples

```
pc$ who --ips
toto      tty7          2011-09-12 15:08 (:0)
toto      pts/1        2011-09-12 15:08 (:0.0)
bidule    2011-09-12 11:12 (139.124.187.124:0)
bidule    pts/2        2011-09-12 11:13 (139.124.187.124:0.0)
root      pts/4        2011-09-12 15:49 (:0.0)
cyril     pts/5        2011-09-12 15:50 (:0.0)
titi      pts/6        2011-09-15 10:06 (172.16.203.1)
$ who -q
toto toto root cyril titi
# usager=5
```

□

Les options **-b** et **-r** donnent des renseignements sur le système : **-b** demande d’afficher l’heure de dernier démarrage de la machine ; **-r** demande d’afficher son niveau d’exécution (*runlevel*).

### Exemple

```
$ who -b
system boot 2011-08-29 14:40
$ who -r
niveau d'exécution 2 2011-08-29 14:40          dernier=S
```

⇒ la machine a démarré en runlevel 2, ce qui correspond, pour la distribution Debian, au mode par défaut (multi-utilisateurs, réseau et mode graphique).

□

## 11.E.2 finger : obtenir des renseignements sur des utilisateurs

La commande **finger** affiche des informations sur des utilisateurs locaux ou distants.

### Synopsis

```
finger [-lps] {utilisateur[@hôte]}
```

D’une certaine manière, **finger** agit comme une sorte d’annuaire et présente des informations utiles et accessibles concernant les utilisateurs en arguments. La forme *utilisateur@hôte* demande des informations sur l’utilisateur de la machine *hôte*. Dans ce cas, **finger** interroge le serveur finger<sup>1</sup> de *hôte* et affiche le résultat. Pour des raisons de sécurité/confidentialité, l’activation du service finger se fait de plus en plus rare.

Les informations accessibles par **finger** pour un utilisateur donné ont diverses origines :

- (de */etc/passwd*) les informations du champ commentaire, comprenant éventuellement : son nom réel, le numéro de son bureau, le numéro de téléphone de son bureau, son numéro personnel et autres ;
- (de */etc/passwd*) son répertoire personnel ;
- (de */etc/passwd*) son interpréteur de commandes par défaut ;
- s’il est connecté : le terminal possédé, la date de connexion, le temps d’inactivité, le *display* ou l’hôte distant ;
- le status de sa boîte mail locale (vide ou date de la dernière lecture) ;
- le contenu des fichiers *.plan*, *.project*, *.forward*, et *.pgpkey* de son répertoire de travail.

1. Un serveur finger, tel que **fingerd**, écoute sur le port 79 de TCP.

Sans option ni argument, **finger** se comporte à peu près comme **who** et affiche la liste des utilisateurs actuellement connectés au système, à raison d'une ligne par utilisateur et par terminal utilisé. Chaque ligne contient le nom d'utilisateur, son nom réel, son terminal, son temps d'inactivité, sa date de connexion, le *display* ou l'hôte distant, le numéro de son bureau et le numéro de téléphone de son bureau.

### Exemple

```
$ finger
Login      Name                Tty      Idle   Login Time   Office      Office Phone
gollar     Golade Larry        *pts/2    3      Sep 20 11:12 (b105.iut.univ-aix.fr)
cpb        cyril pain-barre    pts/1      Sep 20 11:09 (ala02u11.u-3mrs.fr)
leborgne   LEBORGNE DOMINIQUE *pts/0    2:11   Sep 19 09:28 (pcdl.iut.univ-aix.fr)
```

□

L'option **-s** produit le même affichage, éventuellement réduit aux utilisateurs en arguments.

Si au moins un utilisateur est indiqué, ou si l'option **-l** est présente, **finger** présente toutes les informations disponibles en plusieurs lignes pour chaque utilisateur. L'option **-p** demande de ne pas afficher les fichiers `.plan`, `.project` et `.pgpkey`.

### Exemple

```
$ finger -l leborgne cpb
Login: leborgne                Name: LEBORGNE DOMINIQUE
Directory: /users/prof/leborgne Shell: /bin/bash
Office: , 42939042             Home Phone: 0
On since Mon Sep 19 09:28 (CEST) on pts/0 from pcdl.iut.univ-aix.fr
6 hours 40 minutes idle
(messages off)
Mail last read Mon Jul 11 22:58 2011 (CEST)
No Plan.

Login: cpb                     Name: cyril pain-barre
Directory: /users/prof/cpb      Shell: /bin/bash
On since Tue Sep 20 13:23 (CEST) on pts/1 from ala02u11.u-3mrs.fr
Mail forwarded to cyril.pain-barre@univ-amu.fr
No mail.
No Plan.
```

□

## 11.E.3 w : savoir ce que font les utilisateurs sur leurs terminaux

La commande **w** donne des renseignements sur le système, les utilisateurs qui disposent d'un terminal, et sur ce qu'ils y font.

### Synopsis

**w** [-fhs] [utilisateur]

Sans option ni argument, **w** affiche une ligne d'état, une ligne d'en-tête, et pour chaque terminal possédé par un utilisateur, une ligne indiquant ce que l'utilisateur y fait. L'option **-h** (*header*) désactive l'affichage de la ligne d'état et de la ligne d'en-tête.

La ligne d'état contient, dans l'ordre : l'heure système, le temps passé depuis le dernier démarrage, le nombre d'utilisateurs connectés<sup>2</sup>, et la charge moyenne du système lors de la dernière, des 5 dernières et des 15 dernières minutes.

La ligne d'en-tête précise la nature des informations fournies dans les colonnes des lignes qui suivent. Par défaut sur une Debian, l'en-tête a la forme :

```
USER TTY FROM LOGIN@ IDLE JCPU PCPU WHAT
```

où :

- USER : est le nom d'utilisateur ;
- TTY : est le terminal qu'il possède (sans /dev/);
- FROM : indique le nom (ou l'adresse) de l'hôte distant, ou le nom de *display* pour une session graphique ou un affichage déporté ;
- LOGIN@ : date ou heure à laquelle l'utilisateur s'est connecté ;
- IDLE : le temps d'inactivité, autrement dit le temps depuis lequel le terminal est inutilisé ;
- JCPU : la durée d'utilisation de la CPU par les processus rattachés à ce terminal ;
- PCPU : la durée d'utilisation de la CPU par le processus de la colonne WHAT ;
- WHAT : la ligne de commande du processus exécuté en avant-plan sur le terminal.

## Exemples

allegro\$ **w**

```
11:34:40 up 16 days, 20:54, 10 users, load average: 0,09, 0,11, 0,09
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU WHAT
leborgne pts/0     pcdl.iut.univ-ai Mon08    6.00s  0.48s  8.72s sshd: leborgne
ambchr    pts/1     a75.iut.univ-aix 08:18    3:15m  0.10s  0.10s -bash
zekaur    pts/2     a74.iut.univ-aix 08:19    3:13m  0.22s  0.22s -bash
toto      pts/3     a76.iut.univ-aix 08:20    2:50m  0.06s  0.06s -bash
toto      pts/4     a76.iut.univ-aix 11:34   13.00s  0.14s  0.14s -bash
cpb       pts/5     ala02u11.u-3mrs. 09:31    0.00s  0.48s  0.38s w
jamsyl    pts/6     a82.iut.univ-aix 08:44    2:49   0.28s  0.28s -bash
chatom    pts/7     a80.iut.univ-aix 08:44   28.00s  0.14s  0.02s vi TP_ABD_1.sql
toto      pts/9     139.124.214.128 10:44   50:14  0.06s  0.06s -bash
```

⇒ a priori, tous les terminaux actuellement ouverts sur allegro sont de type terminal distant (SSH ou autre).

allegro\$ **w -h**

```
leborgne pts/0     pcdl.iut.univ-ai Mon08    9.00s  0.48s  8.72s sshd: leborgne
ambchr    pts/1     a75.iut.univ-aix 08:18    3:15m  0.10s  0.10s -bash
zekaur    pts/2     a74.iut.univ-aix 08:19    3:13m  0.22s  0.22s -bash
toto      pts/3     a76.iut.univ-aix 08:20    2:50m  0.06s  0.06s -bash
toto      pts/4     a76.iut.univ-aix 11:34   16.00s  0.14s  0.14s -bash
cpb       pts/5     ala02u11.u-3mrs. 09:31    0.00s  0.10s  0.00s w -h
jamsyl    pts/6     a82.iut.univ-aix 08:44    2:52   0.28s  0.28s -bash
chatom    pts/7     a80.iut.univ-aix 08:44    0.00s  0.12s  0.12s -bash
toto      pts/9     139.124.214.128 10:44   50:17  0.06s  0.06s -bash
```

⇒ même résultat sans la ligne d'état ni celle d'en-tête.

2. Ce nombre peut être plus grand que le nombre d'utilisateurs possédant un terminal.



```
pc$ w
16:39:41 up 3 days, 1:31, 12 users, load average: 0,06, 0,07, 0,02
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU WHAT
cyril     tty7      :0              Mon15    3days  1:47m  1.03s x-session-manag
cyril     pts/1     :0.0            Mon15    0.00s  0.71s  0.01s w
bidule    pts/2     139.124.187.124: 11:12    3:51m  2days  0.08s /usr/lib/WindowMa
bidule    pts/2     139.124.187.124: 11:13    50:09   0.18s  0.18s bash
root      pts/4     :0.0            Mon15    2days  0.29s  0.29s -bash
root      pts/8     :0.0            Tue09    6:08m  0.06s  0.00s less /etc/initt
```

⇒ a priori, les terminaux actuellement ouverts sur pc le sont via des sessions graphiques.

□

Si *utilisateur* est fourni en argument, **w** n'affiche que les lignes concernant l'activité de cet *utilisateur*, précédées éventuellement des lignes d'en-tête.

### Exemple

```
allegro$ w toto
11:35:44 up 16 days, 20:55, 8 users, load average: 0,03, 0,09, 0,08
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU WHAT
toto      pts/3     a76.iut.univ-aix 08:20    2:51m  0.06s  0.06s -bash
toto      pts/4     a76.iut.univ-aix 11:34    10.00s  0.15s  0.14s -bash
toto      pts/9     139.124.214.128 10:44    51:18   0.06s  0.06s -bash
```

□

L'option **-f** (*from*) désactive ou active l'affichage de la colonne FROM, selon que **w** a été compilée avec ou sans cet affichage par défaut. Sur Debian, la colonne FROM est affichée par défaut. L'option **-s** (*short*) désactive l'affichage des colonnes LOGIN@, JCPU et PCPU.

### Exemples

```
allegro$ w -f
11:34:57 up 16 days, 20:54, 9 users, load average: 0,07, 0,11, 0,09
USER      TTY      LOGIN@   IDLE   JCPU   PCPU WHAT
leborgne  pts/0     Mon08    23.00s  0.48s  8.72s sshd: leborgne [priv]
ambchr    pts/1     08:18    3:15m  0.10s  0.10s -bash
zekaur    pts/2     08:19    3:13m  0.22s  0.22s -bash
toto      pts/3     08:20    2:50m  0.06s  0.06s -bash
toto      pts/4     11:34    30.00s  0.14s  0.14s -bash
cpb       pts/5     09:31    0.00s  0.10s  0.00s w -f
jamsyl    pts/6     08:44    3:06   0.28s  0.28s -bash
toto      pts/9     10:44    50:31   0.06s  0.06s -bash
```

```
allegro$ w -s
11:34:51 up 16 days, 20:54, 9 users, load average: 0,08, 0,11, 0,09
USER      TTY      FROM            IDLE WHAT
leborgne  pts/0     pcdl.iut.univ-ai 17.00s sshd: leborgne [priv]
ambchr    pts/1     a75.iut.univ-aix 3:15m -bash
zekaur    pts/2     a74.iut.univ-aix 3:13m -bash
toto      pts/3     a76.iut.univ-aix 2:50m -bash
toto      pts/4     a76.iut.univ-aix 24.00s -bash
cpb       pts/5     ala02u11.u-3mrs. 0.00s w -s
jamsyl    pts/6     a82.iut.univ-aix 3:00 -bash
toto      pts/9     139.124.214.128 50:25 -bash
```

□

### 11.E.4 last : connaître les connexions et déconnexions des utilisateurs

La commande **last** permet de connaître les connexions et déconnexions des utilisateurs. Elle donne aussi quelques informations sur les (re)démarrages du système.

#### Synopsis

```
last [-n] [-aix] [-t AAAAMMMJJhhmmss] {utilisateur} {terminal}
```

Sans option ni argument, **last** affiche la liste des connexions et des éventuelles déconnexions des utilisateurs. Pour cela, elle utilise le fichier `/var/log/wtmp` dans lequel sont enregistrées ces informations, ainsi que les informations de (re)démarrage du système. Elle affiche une ligne par connexion d'un utilisateur à un terminal. La ligne comporte dans l'ordre :

- le nom d'utilisateur ;
- le terminal utilisé ;
- le nom (ou à défaut l'adresse) de l'hôte distant, ou éventuellement le *display* (affichage déporté) ;
- la date et l'heure de connexion ;
- la mention "**still logged in**" si l'utilisateur est encore connecté, sinon l'heure de déconnexion suivie de la durée de la connexion. La mention "**gone – no logout**" peut aussi apparaître dans le cas de sessions graphiques improprement terminées, ou avec l'option **-t** (voir plus loin) ;

Aussi, les lignes avec l'utilisateur fictif **reboot** informent sur les redémarrages du système.

Les arguments optionnels *utilisateur* et *terminal* limitent l'affichage aux lignes concernées par les utilisateurs ou les terminaux mentionnés. *utilisateur* peut être aussi **reboot**.

#### Exemples

```
$ last
cpb      pts/1      ala02u11.u-3mrs. Tue Sep 20 09:10    still logged in
bouabd   pts/1      sju13-3-82-234-2 Mon Sep 19 20:22 - 20:28    (00:05)
bouabd   pts/1      sju13-3-82-234-2 Mon Sep 19 20:21 - 20:22    (00:00)
arigui   pts/3      allegro.iut.univ Mon Sep 19 18:51 - 18:52    (00:01)
arigui   pts/2      amarseille-156-1 Mon Sep 19 18:50 - 18:52    (00:02)
...
troseb   pts/1      139.124.214.128  Mon Sep 19 10:23 - 10:26    (00:02)
leborgne pts/0      pcdl.iut.univ-ai Mon Sep 19 09:28    still logged in
cpb      pts/0      ala02u11.u-3mrs. Mon Sep 19 09:25 - 09:25    (00:00)
...
laporte  pts/4      139.124.187.35:0 Thu Sep  8 18:14 - 18:16    (00:01)
laporte  pts/4      139.124.187.35:0 Thu Sep  8 18:14    gone – no logout
reboot   system boot 2.6.32-5-686     Wed Sep  7 08:25 - 15:07    (5+06:42)
...
avijer   pts/1      pcdl.iut.univ-ai Thu Sep  1 17:33 - 17:35    (00:01)
leborgne pts/0      139.124.187.248: Thu Sep  1 10:59    gone – no logout

wtmp begins Thu Sep  1 10:59:34 2011
```

⇒ quelques lignes extraites de la sortie de **last**. La dernière ligne indique la date à laquelle les enregistrements de connexion ont commencé

```
$ last leborgne
```

```
leborgne pts/0      pcdl.iut.univ-ai Mon Sep 19 09:28    still logged in
leborgne pts/1      pcdl.iut.univ-ai Fri Sep 16 10:32 - 10:33    (00:00)
leborgne pts/0      pcdl.iut.univ-ai Thu Sep 15 14:22 - 17:55    (1+03:33)
leborgne pts/2      c133.iut.univ-ai Mon Sep 12 15:05 - 15:17    (00:12)
leborgne pts/0      pcdl.iut.univ-ai Mon Sep 12 08:51 - 14:05    (3+05:13)
leborgne pts/0      ax113-2-82-224-1 Sat Sep 10 16:08 - 16:51    (00:42)
...
```

```
wtmp begins Thu Sep 1 10:59:34 2011
```

⇒ *affichage réduit à un utilisateur en particulier*

```
$ last pts/0
```

```
leborgne pts/0      pcdl.iut.univ-ai Mon Sep 19 09:28    still logged in
cpb      pts/0      ala02u11.u-3mrs. Mon Sep 19 09:25 - 09:25    (00:00)
troseb   pts/0      139.124.214.128 Mon Sep 19 08:38 - 09:09    (00:31)
risch    pts/0      bon13-1-82-232-1 Mon Sep 19 00:55 - 00:56    (00:01)
risch    pts/0      bon13-1-82-232-1 Mon Sep 19 00:49 - 00:52    (00:03)
cbon     pts/0      amarseille-253-1 Sun Sep 18 10:04 - 10:10    (00:05)
...
```

⇒ *affichage réduit à un terminal*

```
$ last leborgne cpb pts/0
```

```
cpb      pts/1      ala02u11.u-3mrs. Tue Sep 20 09:10    still logged in
cpb      pts/1      ala02u11.u-3mrs. Tue Sep 20 09:01 - 09:10    (00:09)
cpb      pts/1      ala02u11.u-3mrs. Mon Sep 19 17:52 - 20:17    (02:25)
cpb      pts/1      ala02u11.u-3mrs. Mon Sep 19 17:38 - 17:39    (00:01)
leborgne pts/0      pcdl.iut.univ-ai Mon Sep 19 09:28    still logged in
cpb      pts/0      ala02u11.u-3mrs. Mon Sep 19 09:25 - 09:25    (00:00)
troseb   pts/0      139.124.214.128 Mon Sep 19 08:38 - 09:09    (00:31)
risch    pts/0      bon13-1-82-232-1 Mon Sep 19 00:55 - 00:56    (00:01)
...
```

⇒ *affiche les lignes concernées par un utilisateur ou un terminal mentionné.*

□

L'option **-n**, où *n* est un entier, demande de limiter l'affichage aux *n* lignes les plus récentes. L'option **-t AAAAMMJJhhmmss** demande de limiter l'affichage aux informations antérieures à la date indiquée par l'argument **AAAAMMJJhhmmss**. Pour les utilisateurs encore connectés à cette date mais déconnecté depuis, la mention **"gone - no logout"** est affichée.

L'option **-a** fait afficher le nom de l'hôte distant en dernière colonne afin qu'il ne soit pas tronqué. L'option **-i** demande à afficher l'adresse IP de connexion plutôt que le nom d'hôte.

Enfin, l'option **-x** demande l'affichage des arrêts du système et des changements de *runlevel*.

## Exemples

```
$ last -a -4 -t 20110908100530
```

```
leborgne pts/1      Thu Sep 8 09:58    gone - no logout  pcdl.iut.univ-aix.fr
cpb      pts/0      Thu Sep 8 09:43    gone - no logout  ala02u11.u-3mrs.fr
leborgne pts/0      Thu Sep 8 08:41 - 08:58    (00:16)  pcdl.iut.univ-aix.fr
zekaur   pts/1      Wed Sep 7 13:26 - 13:26    (00:00)  a70.iut.univ-aix.fr
```

```
wtmp begins Thu Sep 1 10:59:34 2011
```

⇒ affiche les 4 lignes les plus récentes et antérieures au 8 septembre 2011 à 10 heures 5 minutes et 30 secondes

```
$ last -a -4 -t 20110908100530 -i
```

```
leborgne pts/1      Thu Sep  8 09:58      gone - no logout    139.124.187.248
cpb      pts/0      Thu Sep  8 09:43      gone - no logout    195.221.220.107
leborgne pts/0      Thu Sep  8 08:41 - 08:58 (00:16)    139.124.187.248
zekaur   pts/1      Wed Sep  7 13:26 - 13:26 (00:00)    139.124.187.70
```

```
wtmp begins Thu Sep  1 10:59:34 2011
```

⇒ mêmes enregistrements mais affiche les adresses IP plutôt que les noms d'hôte

```
$ last -x
```

```
...
cyril    pts/1      :0.0      Mon Sep 12 15:08 - 09:09 (7+18:00)
cyril    tty7      :0        Mon Sep 12 15:08      still logged in
runlevel (to lvl 2)  2.6.32-5-686 Mon Sep 12 15:07 - 10:27 (7+19:19)
reboot   system boot 2.6.32-5-686 Mon Sep 12 15:07 - 10:27 (7+19:19)
shutdown system down 2.6.32-5-686 Mon Sep 12 15:07 - 15:07 (00:00)
runlevel (to lvl 6)  2.6.32-5-686 Mon Sep 12 15:07 - 15:07 (00:00)
cyril    pts/10     :0.0      Mon Sep 12 11:49 - 14:57 (03:08)
root     pts/9      :0.0      Thu Sep  8 15:59 - 14:57 (3+22:58)
...
runlevel (to lvl 2)  2.6.32-5-686 Wed Sep  7 08:25 - 15:07 (5+06:42)
reboot   system boot 2.6.32-5-686 Wed Sep  7 08:25 - 15:07 (5+06:42)
shutdown system down 2.6.32-5-686 Wed Sep  7 07:36 - 08:25 (00:49)
runlevel (to lvl 0)  2.6.32-5-686 Wed Sep  7 07:35 - 07:36 (00:00)
...
```

⇒ (sur un autre ordinateur) l'option **-x** fait apparaître les arrêts et les changements de runlevel : le passage au niveau 2 est un démarrage normal, au niveau 6 c'est le redémarrage, et au niveau 0 c'est l'arrêt.

□

# Chapitre 12

## Expressions régulières et utilitaires

### 12.A Expressions régulières

Les **expressions régulières** (appelées aussi **expressions rationnelles**) est un terme désignant l'expression de motifs de chaînes de caractères. Ceci, dans l'esprit des motifs de noms de fichiers, en beaucoup plus puissant. Elles servent à la recherche et/ou au remplacement de chaînes dans des lignes de texte et sont reconnues par de nombreux utilitaires d'Unix tels que **vi**, **grep**, **more**, **less**, **man**, **sed**,...

Leur introduction dans les utilitaires Unix est très ancienne et a largement contribué à rendre populaire le système auprès des administrateurs système.



**Les expressions régulières ne sont pas destinées à être traitées par bash mais par certains utilitaires.**

Une expression régulière est formée de motifs exprimés par des suites de caractères. Les caractères **^**, **.**, **\$**, **[**, **]**, et **\*** sont spéciaux dans une expression régulière.



Ces caractères (à part **.**) sont aussi des caractères spéciaux pour bash et il faudra faire en sorte qu'il ne les interprète pas.

Les caractères spéciaux ont une signification différente de leur symbole. Pour qu'un caractère spécial soit interprété littéralement (comme son propre symbole), il faut le faire précéder d'un *backslash* **\**.

Les caractères suivants sont normaux mais acquièrent une signification spéciale s'ils sont précédés d'un *backslash* <sup>1</sup> : **?**, **+**, **{**, **}**, **(**, **)** et **|**. Pour que le *backslash* soit reconnu comme lui-même, il suffit de le doubler. Tous les autres caractères ne représentent qu'eux-mêmes. En particulier, tout espace présent dans l'expression régulière doit figurer aussi dans la chaîne.

#### Syntaxe d'une expression régulière (deux définitions possibles) :


`exp_reg ::= [ ^ ] motif [ modificateur ] { motif [ modificateur ] } [ $ ]`

*motif* est une description incomplète d'une partie d'une chaîne. Si **^** est présent en début du premier *motif*,

---

1. Certains utilitaires admettent un mode d'expressions régulières étendues dans lequel ces caractères n'ont pas besoin d'être précédés d'un *backslash* pour être spéciaux. Par exemple, pour **grep** l'option **-E** permet d'activer ce mode. Pour **sed**, il s'agit de l'option **-r**.

alors la chaîne doit se trouver au début d'une ligne. Si **\$** est présent en fin du dernier *motif*, elle doit se trouver à la fin de la ligne ;


 **^** est interprété littéralement (perd sa signification particulière) s'il n'est pas placé en début d'une expression régulière. De même, **\$** est interprété littéralement s'il n'est pas placé en fin d'une expression régulière. Ainsi qu'il a été dit précédemment, ils sont interprétés littéralement s'ils sont précédés d'un **\**.

$exp\_reg ::= exp\_reg \backslash | exp\_reg$   
définit une alternative entre deux expressions régulières.

### L'expression de motif doit suivre la syntaxe suivante :

$motif ::= .$   
le point remplace n'importe quel caractère ;

$motif ::= [ [ ^ ] ensemble ]$   
c'est la même définition d'un *ensemble* pour les motifs de noms de fichiers. *ensemble* peut contenir des intervalles de caractères. Par exemple, **a-z** représente l'ensemble des lettres minuscules **non accentuées**<sup>2</sup>. Les caractères spéciaux perdent leur signification dans l'expression de *ensemble*.

 Pour faire figurer le caractère **]** dans *ensemble*, il faut le placer en premier. Pour faire figurer **^**, il ne faut pas le placer en premier. Pour faire figurer un **-**, il faut le placer en dernier.

$motif ::= \backslash w$   
un caractère alphanumérique (lettre non accentuée quelle que soit sa casse, ou chiffre) ;

$motif ::= \backslash W$   
n'importe quel caractère non alphanumérique ;

$motif ::= caract\grave{e}re\_normal$   
un caractère normal ne représente que lui-même ;

$motif ::= \backslash (exp\_reg \backslash )$   
permet de grouper une expression régulière (en vue éventuellement d'utiliser un modificateur). De plus, la partie de texte qui a été reconnue comme correspondant à l'expression entre parenthèses est mémorisée et peut être réutilisée (voir ci-dessous) ;

$motif ::= \backslash n$   
où  $1 \leq n \leq 9$ . Remplacé par la partie de texte qui correspondait à la *n*ème expression entre parenthèses. Par exemple, si la première expression entre parenthèses est **\(a\*\)** et qu'elle correspond à **aaa** dans la ligne analysée, alors **\1** représente (pour cette ligne) **aaa**.

2. Selon le jeu de caractères utilisé (ou plus exactement ce qu'on appelle la *locale*), il se peut que les caractères soient classés en ne tenant pas compte de la casse. Dans ce cas, **[a-d]** pourrait être interprété comme **[aBbCcDd]** (sans **A**). Il existe des solutions pour régler ce problème (modification de la variable **LC\_ALL**, classes de caractères). Consulter le manuel de **grep** pour plus d'explications.

## Les modificateurs

Ils permettent de fixer des contraintes sur le nombre d'apparitions du *motif* (ou donc de l'expression régulière entre parenthèses) qui le précède. Un *modificateur* s'exprime selon la syntaxe suivante :

*modificateur* ::= \*

le motif précédent peut apparaître 0 ou plus d'une fois (nombre indéfini) ;

*modificateur* ::= \+

le motif précédent doit apparaître au moins une fois (et peut être plus) ;

*modificateur* ::= \?

le motif précédent doit apparaître au plus une fois (peut être 0) ;

*modificateur* ::= \{ *n* [, *m*] \}

le motif précédent doit apparaître *n* fois, ou entre *n* et *m* fois si , *m* est spécifié ;

*modificateur* ::= \{ *n* , \}

le motif précédent doit apparaître au moins *n* fois.

## Les motifs fictifs

À l'instar de ^ et de \$ qui ne représentent aucun caractère s'ils sont placés respectivement en début et en fin d'une expression régulière, il existe des motifs fictifs **qui ne représentent aucun caractère** mais peuvent s'avérer utiles pour situer des (parties) d'expressions :

\< correspond à un début de mot ;

\> correspond à une fin de mot ;

\b correspond à un début ou une fin de mot ;

\B correspond à ce qui n'est ni un début, ni une fin de mot.

❗ Ce sont les chaînes les plus grandes correspondant au *motif* modifié qui seront retenues. Par exemple, le motif **a\*** aura pour correspondance **aaa** dans la chaîne **bbbaaaccc** et non pas la chaîne vide.  
(à méditer)

## 12.B Utilitaires majeurs traitant les expressions régulières

### 12.B.1 grep : filtrer les lignes de fichiers

La commande externe **grep** (*global regular expression parser*) est un filtre des lignes des fichiers en arguments ou de l'entrée standard. Elle n'affiche que les lignes contenant une chaîne correspondant à une expression régulière.

#### Synopsis

**grep** [-vclniwrEq] *exp\_reg* { *référence* }

*référence* est une référence à un fichier ordinaire. Si aucune *référence* n'est spécifiée, **grep** ne filtre que les lignes lues sur l'entrée standard. L'entrée standard est aussi lue si une *référence* est –.

Sans option, les lignes contenant une chaîne correspondant à *exp\_reg* sont affichées (écrites sur la sortie standard). Si plusieurs *références* sont spécifiées, chaque ligne affichée est précédée du nom du fichier qui la contient suivi de :.

L'option **-v** inverse la recherche et affiche celles ne contenant pas de chaîne correspondant à *exp\_reg*.


L'option **-c** fait simplement afficher le nombre de lignes qui auraient été écrites (pour chaque fichier).

L'option **-l** permet de n'afficher que les noms des fichiers dont au moins une ligne correspond à *exp\_reg*.

L'option **-n** demande à ce que chaque ligne écrite soit précédée de son numéro d'apparition dans le fichier.

L'option **-i** demande de ne pas distinguer les majuscules et les minuscules.

L'option **-w** indique que *exp\_reg* doit correspondre à un mot.

 Dans le contexte des expressions régulières et utilitaires associés, un « mot » est constitué de caractères alphabétiques, de chiffres et du caractère souligné. Une chaîne est reconnue comme un mot si elle est encadrée par des caractères différents de ceux là (modulo son emplacement éventuel en début ou en fin de ligne).

L'option **-r** demande d'effectuer la recherche dans tous les fichiers de l'arborescence des *références* qui sont des répertoires.

L'option **-E** active les expressions régulières étendues.

L'option **-q** demande de ne rien afficher, même pas les erreurs, et de se terminer avec une valeur de retour<sup>3</sup> à 0 dès qu'une concordance est trouvée.

## Exemples

```
$ cat fic1
grep est un utilitaire
qui n'est pas inutile
bien au contraire

utile
et franchement pas futile

aussi fut-il largement deploye.
il est
utilise sur divers
systemes...
```

3. Les valeurs de retour seront vues dans la section 16.B.1 page 237.



```
$ grep util fic1
```

```
grep est un utilitaire
qui n'est pas inutile
utile
et franchement pas futile
utilise sur divers
```

⇒ affiche les lignes qui contiennent la suite de caractères **util**

```
$ grep utile fic1
```

```
qui n'est pas inutile
utile
et franchement pas futile
```

⇒ affiche les lignes qui contiennent la suite de caractères **utile**

```
$ grep 'ut.\?il' fic1
```

```
grep est un utilitaire
qui n'est pas inutile
utile
et franchement pas futile
aussi fut-il largement deploye.
utilise sur divers
```

⇒ affiche les lignes qui contiennent la suite de caractères **ut** suivi éventuellement d'un caractère puis de **il**. On remarque que la ligne contenant *fut-il* correspond.

```
$ grep -wn utile fic1
```

```
5:utile
```

⇒ affiche le numéro et les lignes qui contiennent le mot **utile**

```
$ grep '^util' fic1
```

```
utile
utilise sur divers
```

⇒ lignes qui commencent par **util**

```
$ grep 'utile$' fic1
```

```
qui n'est pas inutile
utile
et franchement pas futile
```

⇒ lignes qui se terminent par **utile**

```
$ grep -n '^_*$' fic1
```

```
4:
7:
```

⇒ numéro et contenu des lignes blanches (ne contenant que des espaces ou rien)

```
$ grep -c '^_*$' fic1
```

```
2
```

⇒ compte les lignes blanches

```
$ grep -v '^_*$' fic1
```

```
grep est un utilitaire
qui n'est pas inutile
bien au contraire
utile
```

```
et franchement pas futile
aussi fut-il largement deploye.
il est
utilise sur divers
systemes...
```

⇒ *tout sauf les lignes blanches*

```
$ grep '\.' fic1
aussi fut-il largement deploye.
systemes...
```

⇒ *lignes contenant le caractère . (point)*

```
$ grep '^[aeiouy]' fic1
utile
et franchement pas futile
aussi fut-il largement deploye.
il est
utilise sur divers
```

⇒ *lignes commençant par une voyelle*

```
$ grep -v '^[aeiouy]' fic1
grep est un utilitaire
qui n'est pas inutile
bien au contraire
```

```
systemes...
```

⇒ *lignes commençant par autre chose qu'une voyelle. On remarque que les lignes blanches sont affichées. En effet, l'option -v demande d'afficher toutes les lignes qui ne correspondent pas à l'expression régulière.*

```
$ grep -v '^[aeiouy]' fic1 | grep -v '^_*$'
grep est un utilitaire
qui n'est pas inutile
bien au contraire
systemes...
```

⇒ *élimine de la sortie précédente les lignes blanches*

```
$ grep '^^[aeiouy]' fic1
grep est un utilitaire
qui n'est pas inutile
bien au contraire
systemes...
```

⇒ *lignes commençant par un caractère différent d'une voyelle. Les lignes blanches ne sont pas affichées car elles sont vides. Elles l'auraient été si elles contenaient au moins un blanc (même une tabulation).*

```
$ grep '^[a-e]' fic1
bien au contraire
et franchement pas futile
aussi fut-il largement deploye.
```

⇒ *lignes qui commencent par un caractère entre a et e*

```
$ grep '^[a-e].*[^aeiouy]$' fic1
```

aussi fut-il largement deploye.

⇒ lignes qui commencent par un caractère entre **a** et **e** et qui se terminent par autre chose qu'une voyelle

```
$ grep '\([a-z]\+[a-z]\+\)\{4,\}[a-z]\+' fic1
```

qui n'est pas inutile

aussi fut-il largement deploye.

⇒ lignes qui contiennent au moins 5 mots (au moins 4 fois une suite de caractères alphabétiques suivie d'une suite de caractères non alphabétiques, le tout suivi d'une suite de caractères alphabétiques)

```
$ grep '^(\.\\)\{2,\}' fic1
```

et franchement pas futile

systemes...

⇒ lignes dont le premier caractère apparaît au moins trois fois (se répète au moins deux fois)

```
$ grep '^(\.\\).*\1$' fic1
```

et franchement pas futile

⇒ lignes qui commencent et se terminent par le même caractère

```
$ grep '^(\.\\).*\1\.*$' fic1
```

et franchement pas futile

systemes...

⇒ lignes qui commencent et se terminent par le même caractère en ignorant les points éventuels en fin de ligne

□

## 12.B.2 sed : éditer un flux de texte

L'utilitaire **sed** (*stream editor*) est l'un des plus anciens d'Unix. Son rôle est d'écrire sur sa sortie standard la transformation des lignes lues dans des fichiers ou l'entrée standard. Il s'avère plutôt utile lorsqu'on veut automatiser des transformations de texte.

### Synopsis

```
sed [-ns] [-i[suffixe]] {-e script} {-f fichier-script} {référence}
```

Lit les lignes des fichiers *référence*, ou à défaut l'entrée standard, et écrit le résultat de leur transformation par les *scripts* indiqués. Si une *référence* est **-**, **sed** traite aussi l'entrée standard.

Les transformations à opérer sont indiquées par des *commandes* dont on peut indiquer la *portée* par une adresse, voire un intervalle d'adresses. Un script contient un ensemble de commandes (avec leur portée). L'option **-e** ajoute les commandes figurant dans la *chaîne script*. L'option **-f** ajoute les commandes contenues dans le *fichier fichier-script*.

Par défaut, **sed** ne modifie pas les fichiers traités, elle se contente d'écrire sur la sortie standard le résultat de son traitement. L'option **-i** demande à ce que **sed** les modifie, sans utiliser sa sortie standard<sup>4</sup>. L'option **-i** accepte un argument optionnel *suffixe* lui demandant de faire une sauvegarde du fichier *référence* avant de le modifier. Le fichier sauvegardé porte le nom du fichier traité, suivi de *suffixe*.

4. En réalité, **sed** utilise des fichiers temporaires qu'il renomme ensuite à la place des fichiers d'origine

**sed** opère par cycle : lecture d'une ligne, traitement des commandes, écriture éventuelle de la ligne transformée, lecture de la ligne suivante, etc. Au départ, la ligne est placée dans un tampon qui va être modifié par certaines commandes. Les commandes sont traitées selon leur ordre d'apparition sur la ligne de commandes ou dans les fichiers scripts. Si la portée de la commande correspond au tampon, elle est appliquée.

Les commandes **s** et **y** modifient le tampon. D'autres (comme **=**, **a**, **i** et **c**) n'affectent pas le tampon mais ont un effet sur la sortie (lignes en plus ou remplacées). Les modifications qu'elles apportent ne peuvent donc pas être affectées par les commandes qui les suivent.

Lorsque toutes les commandes ont été traitées et si la commande **c** ne s'applique pas, le tampon est écrit sur la sortie standard (sauf si option **-i**). L'option **-n** empêche l'écriture du tampon qui doit être explicitement demandée par la commande **p** (voir aussi option **p** de la commande **s**) mais n'empêche pas l'écriture des lignes ajoutées/remplacées par les commandes **=**, **a**, **i** et **c**.

Par défaut, **sed** traite l'ensemble des entrées comme un unique fichier. Cela a une importance sur la portée des commandes car les adresses (numéros) des lignes lues ne sont pas réinitialisées entre chaque entrée. L'option **-s** fait traiter chaque entrée individuellement, et réinitialiser les numéros de ligne. La première ligne de chaque entrée portera alors le numéro 1. Notons que l'option **-i** active automatiquement l'option **-s**.

De nombreuses commandes sont disponibles. On se limitera à un petit extrait dont les plus utiles sont certainement les blocs, et les commandes **s** et **y**.

Les scripts contiennent plusieurs commandes (aller à la ligne entre chaque commande). La portée de la commande doit la précéder. Le format général est le suivant :


[ *portée* ] [ **!** ] *commande*

### Portée des commandes :

Lorsqu'on ne spécifie pas de portée pour une commande, elle est appliquée sur toutes les lignes. La portée peut être une adresse ou un intervalle.

- une adresse est exprimée par :

<i>n</i>	pour indiquer la ligne numéro <i>n</i> ;
<b>\$</b>	pour indiquer la dernière ligne ;
<i>début~pas</i>	pour indiquer toutes les <i>pas</i> lignes à partir de la ligne numéro <i>début</i> (i.e. <i>début</i> , <i>début+pas</i> , etc.) ;
<i>/exp_reg/</i> [ <b>I</b> ]	pour les lignes contenant une chaîne correspondant à l'expression régulière <i>exp_reg</i> . L'option <b>I</b> demande ne pas tenir compte de la casse.

 Ainsi, une adresse ne désigne pas toujours une seule ligne mais peut en désigner plusieurs.

- un intervalle a la forme *adresse<sub>1</sub>*, *adresse<sub>2</sub>*. Il représente le bloc de lignes commençant à la première correspondant à *adresse<sub>1</sub>* et se terminant à la première correspondant à *adresse<sub>2</sub>* (inclusive). Si *adresse<sub>1</sub>* est une expression régulière, outre ce bloc de lignes, celles correspondant à *adresse<sub>1</sub>* sont aussi retenues. Si *adresse<sub>2</sub>* est aussi une expression régulière, alors l'intervalle représente plusieurs blocs de lignes : les blocs qui commencent par une ligne correspondant à *adresse<sub>1</sub>* et se terminent par une ligne correspondant à *adresse<sub>2</sub>*, puis un éventuel bloc qui commence à une ligne correspondant à *adresse<sub>1</sub>* jusqu'à la fin du fichier.

Si *portée* est suivie du caractère **!**, alors son sens est opposé : les lignes retenues seront celles ne correspondant pas à la portée.

### Commandes avec éventuellement une adresse :

<b>q</b> [ <i>valeur</i> ]	( <i>quit</i> ) ignorer les commandes suivantes, écrire le résultat courant du traitement du tampon (le tampon n'est pas écrit si option <b>-n</b> ) puis arrêter <b>sed</b> avec <i>valeur</i> pour valeur de retour <sup>5</sup> ;
<b>Q</b> [ <i>valeur</i> ]	( <i>quit</i> ) terminer immédiatement avec <i>valeur</i> pour valeur de retour ;
<b>=</b>	faire précéder l'écriture éventuelle du tampon par une ligne contenant le numéro de ligne ;
<b>a</b> \ <i>texte</i>	( <i>append</i> ) faire suivre l'écriture éventuelle du tampon par une ligne contenant <i>texte</i> . <i>texte</i> peut contenir plusieurs lignes si toutes, sauf la dernière, se terminent par un <b>\</b>
<b>i</b> \ <i>texte</i>	( <i>insert</i> ) faire précéder l'écriture éventuelle du tampon par une ligne contenant <i>texte</i> (même format que pour <b>a</b> ).

### Commandes acceptant un intervalle :

**s** / *exp\_reg* / *chaîne\_remp* / [ *options* ] (*substitute*) modifie le tampon en remplaçant la chaîne correspondant à l'expression régulière *exp\_reg* par *chaîne\_remp*. C'est certainement la plus utile et la plus complexe des commandes **sed**. Notons que selon la version, toutes les constructions des expressions régulières ne sont pas reconnues.

*chaîne\_remp* n'est pas qu'une simple chaîne. Elle peut contenir certaines constructions spéciales : **&** représente la chaîne qui correspond à *exp\_reg* ; **\n** représente la chaîne correspondant au *n*<sup>e</sup> groupement de parenthèses ; et d'autres possibilités (**\L**, **\l**, **\U**, **\u** et **\E**) qui ne sont pas développées ici<sup>6</sup>.

De nombreuses options existent dont **g** (*global*), **n** (*number*), **i** (*ignore case*) et **p** (*print*). Sans l'option **g**, seule la première chaîne (du tampon) correspondant à *exp\_reg* est remplacée. Avec l'option **g**, toutes les chaînes qui correspondent à *exp\_reg* le sont. L'option **n**, où *n* est un nombre, fait que seule la *n*<sup>e</sup> chaîne correspondant à *exp\_reg* est remplacée. La combinaison **gn** veut dire de la *n*<sup>e</sup> à la dernière. L'option **i** demande d'ignorer la casse. L'option **p** demande l'affichage du tampon si celui-ci a été modifié par la substitution (utile avec l'option globale **-n**) ;

{ *commandes* } (*block*) construit un bloc de commandes (une par ligne). Particulièrement utile lorsque plusieurs commandes ont la même portée ;

**y** / *source* / *dest* / modifie le tampon en remplaçant chaque caractère de *source* par le caractère correspondant de *dest*. *source* et *dest* doivent contenir le même nombre de caractères.

**p** (*print*) écrire le tampon (utile avec l'option **-n**) ;

**c** \  
*texte* remplace les lignes sélectionnées par *texte* (même format que pour **a**). Puisque le tampon est supprimé, a pour effet de commencer un nouveau cycle (les commandes qui suivent sont ignorées) ;

5. Les valeurs de retour seront vues section 16.B.1 page 237.

6. Voir la documentation complète avec **info sed** (**man** est insuffisant).

## Exemples

```
$ ls -l
total 24
-rw-r--r-- 1 bart simpson 27 jan 12 12:26 avant.sed
-rw-r--r-- 1 bart simpson 0 jan 12 12:01 fic1.txt
-rw-r--r-- 1 bart simpson 0 jan 12 12:01 fic2
-rw-r--r-- 1 bart simpson 0 jan 12 12:06 FICH1.TXT
-rw-r--r-- 1 bart simpson 0 jan 12 12:01 fichier1.uu
-rw-r--r-- 1 bart simpson 0 jan 12 12:02 fic.txt.uu
-rw-r--r-- 1 bart simpson 57 jan 12 12:16 maj.sed
-rw-r--r-- 1 bart simpson 57 jan 12 12:16 min.sed
drwxr-xr-x 2 bart simpson 4096 jan 12 12:07 Rep
drwxr-xr-x 2 bart simpson 4096 jan 12 12:01 repl
drwxr-xr-x 2 bart simpson 4096 jan 12 12:02 repertoire1
```

On va réaliser des transformations sur le résultat de **ls -l** :

```
$ ls -l | sed -e '/sed/='
total 24
2
-rw-r--r-- 1 bart simpson 27 jan 12 12:26 avant.sed
-rw-r--r-- 1 bart simpson 0 jan 12 12:01 fic1.txt
-rw-r--r-- 1 bart simpson 0 jan 12 12:01 fic2
-rw-r--r-- 1 bart simpson 0 jan 12 12:06 FICH1.TXT
-rw-r--r-- 1 bart simpson 0 jan 12 12:01 fichier1.uu
-rw-r--r-- 1 bart simpson 0 jan 12 12:02 fic.txt.uu
8
-rw-r--r-- 1 bart simpson 57 jan 12 12:16 maj.sed
9
-rw-r--r-- 1 bart simpson 57 jan 12 12:16 min.sed
drwxr-xr-x 2 bart simpson 4096 jan 12 12:07 Rep
drwxr-xr-x 2 bart simpson 4096 jan 12 12:01 repl
drwxr-xr-x 2 bart simpson 4096 jan 12 12:02 repertoire1
```

⇨ Demande l’affichage du numéro de ligne avant toutes les lignes contenant *sed*

```
$ ls -l | sed -n -e '4~3p'
-rw-r--r-- 1 bart simpson 0 jan 12 12:01 fic2
-rw-r--r-- 1 bart simpson 0 jan 12 12:02 fic.txt.uu
drwxr-xr-x 2 bart simpson 4096 jan 12 12:07 Rep
```

⇨ N’affiche que les lignes 4, 7, 10, etc.

```
$ ls -l | sed -e 's/bart_/homer/'
total 24
-rw-r--r-- 1 homer simpson 27 jan 12 12:26 avant.sed
-rw-r--r-- 1 homer simpson 0 jan 12 12:01 fic1.txt
-rw-r--r-- 1 homer simpson 0 jan 12 12:01 fic2
-rw-r--r-- 1 homer simpson 0 jan 12 12:06 FICH1.TXT
-rw-r--r-- 1 homer simpson 0 jan 12 12:01 fichier1.uu
-rw-r--r-- 1 homer simpson 0 jan 12 12:02 fic.txt.uu
-rw-r--r-- 1 homer simpson 57 jan 12 12:16 maj.sed
-rw-r--r-- 1 homer simpson 57 jan 12 12:16 min.sed
drwxr-xr-x 2 homer simpson 4096 jan 12 12:07 Rep
drwxr-xr-x 2 homer simpson 4096 jan 12 12:01 repl
drwxr-xr-x 2 homer simpson 4096 jan 12 12:02 repertoire1
```

⇒ Remplace `bart_` par `homer` sur toutes les lignes lues

```
$ ls -l | sed -e '/u$/,/p$/s/bart_/homer/'
total 24
-rw-r--r-- 1 bart      simpson      27 jan 12 12:26 avant.sed
-rw-r--r-- 1 bart      simpson      0 jan 12 12:01 fic1.txt
-rw-r--r-- 1 bart      simpson      0 jan 12 12:01 fic2
-rw-r--r-- 1 bart      simpson      0 jan 12 12:06 FICH1.TXT
-rw-r--r-- 1 homer     simpson      0 jan 12 12:01 fichier1.uu
-rw-r--r-- 1 homer     simpson      0 jan 12 12:02 fic.txt.uu
-rw-r--r-- 1 homer     simpson      57 jan 12 12:16 maj.sed
-rw-r--r-- 1 homer     simpson      57 jan 12 12:16 min.sed
drwxr-xr-x 2 homer     simpson    4096 jan 12 12:07 Rep
drwxr-xr-x 2 bart      simpson    4096 jan 12 12:01 repl
drwxr-xr-x 2 bart      simpson    4096 jan 12 12:02 repertoire1
```

⇒ Traitement des blocs de lignes dont la première ligne finit par `u` et dont la dernière ligne finit par `p` ou est la fin de fichier. Pour ces blocs, remplace `bart_` par `homer`

```
$ ls -l | sed -e 's/^d.*$/&_(répertoire)/'
total 24
-rw-r--r-- 1 bart      simpson      27 jan 12 12:26 avant.sed
-rw-r--r-- 1 bart      simpson      0 jan 12 12:01 fic1.txt
-rw-r--r-- 1 bart      simpson      0 jan 12 12:01 fic2
-rw-r--r-- 1 bart      simpson      0 jan 12 12:06 FICH1.TXT
-rw-r--r-- 1 bart      simpson      0 jan 12 12:01 fichier1.uu
-rw-r--r-- 1 bart      simpson      0 jan 12 12:02 fic.txt.uu
-rw-r--r-- 1 bart      simpson      57 jan 12 12:16 maj.sed
-rw-r--r-- 1 bart      simpson      57 jan 12 12:16 min.sed
drwxr-xr-x 2 bart      simpson    4096 jan 12 12:07 Rep (répertoire)
drwxr-xr-x 2 bart      simpson    4096 jan 12 12:01 repl (répertoire)
drwxr-xr-x 2 bart      simpson    4096 jan 12 12:02 repertoire1 (répertoire)
```

⇒ Pour toutes les lignes qui commencent par `d` (donc concernant un répertoire), rajouter `_` (répertoire) à la fin de la ligne. On voit ici une utilisation de `&`.

```
$ ls -l | sed -n -e '/^d/s/$/_ (répertoire)/p'
drwxr-xr-x 2 bart      simpson    4096 jan 12 12:07 Rep (répertoire)
drwxr-xr-x 2 bart      simpson    4096 jan 12 12:01 repl (répertoire)
drwxr-xr-x 2 bart      simpson    4096 jan 12 12:02 repertoire1 (répertoire)
```

⇒ En combinant l'option `-n`, la commande `s` et son option `p`, on ne garde que les lignes concernant un répertoire, sur lesquelles on effectue une transformation.

```
$ cat maj.sed
y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
$ ls -l | sed -f maj.sed
TOTAL 24
-RW-R--R-- 1 BART      SIMPSON      27 JAN 12 12:26 AVANT.SED
-RW-R--R-- 1 BART      SIMPSON      0 JAN 12 12:01 FIC1.TXT
-RW-R--R-- 1 BART      SIMPSON      0 JAN 12 12:01 FIC2
-RW-R--R-- 1 BART      SIMPSON      0 JAN 12 12:06 FICH1.TXT
-RW-R--R-- 1 BART      SIMPSON      0 JAN 12 12:01 FICHER1.UU
-RW-R--R-- 1 BART      SIMPSON      0 JAN 12 12:02 FIC.TXT.UU
-RW-R--R-- 1 BART      SIMPSON      57 JAN 12 12:16 MAJ.SED
-RW-R--R-- 1 BART      SIMPSON      57 JAN 12 12:16 MIN.SED
DRWXR-XR-X 2 BART      SIMPSON    4096 JAN 12 12:07 REP
```

```
DRWXR-XR-X      2 BART      SIMPSON      4096 JAN 12 12:01 REP1
DRWXR-XR-X      2 BART      SIMPSON      4096 JAN 12 12:02 REPERTOIRE1
```

⇒ Remplace les minuscules par des majuscules. On aurait pu utiliser la commande **y** directement sur la ligne de commandes (avec l'option **-e**) et non dans un script (option **-f**).

```
$ ls -l | sed -f maj.sed -e 'y/ABCDEFGHIJKLMNOPQRSTUVWXYZ/abcdefghijklmnopqrstuvwxyz/'
total 24
-rw-r--r--      1 bart      simpson      27 jan 12 12:26 avant.sed
-rw-r--r--      1 bart      simpson          0 jan 12 12:01 fic1.txt
-rw-r--r--      1 bart      simpson          0 jan 12 12:01 fic2
-rw-r--r--      1 bart      simpson          0 jan 12 12:06 fich1.txt
-rw-r--r--      1 bart      simpson          0 jan 12 12:01 fichier1.uu
-rw-r--r--      1 bart      simpson          0 jan 12 12:02 fic.txt.uu
-rw-r--r--      1 bart      simpson      57 jan 12 12:16 maj.sed
-rw-r--r--      1 bart      simpson      57 jan 12 12:16 min.sed
drwxr-xr-x      2 bart      simpson     4096 jan 12 12:07 rep
drwxr-xr-x      2 bart      simpson     4096 jan 12 12:01 repl
drwxr-xr-x      2 bart      simpson     4096 jan 12 12:02 repertoire1
```

⇒ Remplace les majuscules par des minuscules **après** avoir remplacé les minuscules par des majuscules

```
$ cat avant.sed
1~5i\
GROUPE DE 5 LIGNES :
$ ls -l | sed -f avant.sed
GROUPE DE 5 LIGNES :
total 24
-rw-r--r--      1 bart      simpson      27 jan 12 12:26 avant.sed
-rw-r--r--      1 bart      simpson          0 jan 12 12:01 fic1.txt
-rw-r--r--      1 bart      simpson          0 jan 12 12:01 fic2
-rw-r--r--      1 bart      simpson          0 jan 12 12:06 FICH1.TXT
GROUPE DE 5 LIGNES :
-rw-r--r--      1 bart      simpson          0 jan 12 12:01 fichier1.uu
-rw-r--r--      1 bart      simpson          0 jan 12 12:02 fic.txt.uu
-rw-r--r--      1 bart      simpson      57 jan 12 12:16 maj.sed
-rw-r--r--      1 bart      simpson      57 jan 12 12:16 min.sed
drwxr-xr-x      2 bart      simpson     4096 jan 12 12:07 Rep
GROUPE DE 5 LIGNES :
drwxr-xr-x      2 bart      simpson     4096 jan 12 12:01 repl
drwxr-xr-x      2 bart      simpson     4096 jan 12 12:02 repertoire1
```

⇒ devant chaque bloc de 5 lignes, écrire un message.

```
ls -l | sed -n -e '3,7{
> y/r/R/
> p }'
-Rw-R--R--      1 baRt      simpson          0 jan 12 12:01 fic1.txt
-Rw-R--R--      1 baRt      simpson          0 jan 12 12:01 fic2
-Rw-R--R--      1 baRt      simpson          0 jan 12 12:06 FICH1.TXT
-Rw-R--R--      1 baRt      simpson          0 jan 12 12:01 fichier1.uu
-Rw-R--R--      1 baRt      simpson          0 jan 12 12:02 fic.txt.uu
```

⇒ n'écrire que les lignes 3 à 7 en remplaçant **r** par **R**

□



### 12.B.3 awk : programmer le traitement de fichiers

AWK est un langage de programmation très ancien et encore beaucoup utilisé, en particulier par les administrateurs système. La commande externe **awk** interprète un programme AWK pour traiter un ou plusieurs fichiers. Elle peut agir comme un filtre (de type **grep**), un traitement de flux (de type **sed**), mais peut aussi opérer des traitements plus complexes, grâce aux possibilités offertes par son langage, notamment les fonctions et les structures de contrôle (instructions conditionnelles et répétitives). Néanmoins, nous nous limiterons à une description très sommaire.

Pour fonctionner, **awk** a besoin d'un programme AWK et de fichiers à traiter en entrée.

#### Synopsis

```
awk texte_programme { fic_entrée }
awk -f fichier_programme { fic_entrée }
```

Les fichiers d'entrée sont traités dans l'ordre d'apparition sur la ligne de commande. S'il n'y a pas de *fic\_entrée*, les entrées sont lues dans l'entrée standard. Si un *fic\_entrée* est `-`, l'entrée standard est lue.

Le programme AWK peut être fourni directement sur la ligne de commande par le **mot**<sup>7</sup> *texte\_programme*.

#### Exemple

```
$ cat fic1
1a 1b 1c
2a 2b 2c
3a 3b 3c
$ awk '{ print $1 }' fic1
1a
2a
3a
```

⇒ Affiche le premier mot de chaque ligne. Ce programme sera expliqué un peu plus tard.

□

Le programme AWK peut être contenu dans un fichier, auquel cas il faut utiliser l'option **-f** *fichier\_programme*.

#### Exemple

```
$ cat prog1.awk
{
    print $1
}
```

⇒ dans un fichier, on peut plus facilement aérer l'écriture du programme...

```
$ awk -f prog1.awk fic1
1a
2a
3a
```

□

Enfin, une autre façon très pratique d'exécuter un fichier contenant un programme AWK est d'y **placer au début du fichier, la ligne shebang**<sup>8</sup> suivante :

7. Dans ce cas, on encadre généralement le texte du programme par des quotes (plus rarement par des guillemets) pour ne former qu'un seul mot et éviter l'interprétation des caractères spéciaux par le shell.

8. La ligne shebang est présentée en même temps que l'exécution de scripts bash.

```
#!/usr/bin/awk -f
```

puis de rendre le fichier exécutable et de l'exécuter directement.

### Exemple

```
$ cat prog2.awk
#!/usr/bin/awk -f

{
    print $1
}
$ chmod u+x prog2.awk
$ ./prog2.awk fic1
1a
2a
3a
```

□

## Structure d'un programme AWK et déroulement

Un programme AWK est une liste<sup>9</sup> d'**actions** dépendant chacune d'une éventuelle **condition** :

```
{ [condition] { action } }
```

Sans option particulière, le principe général est le suivant : pour chaque ligne du(des) fichier(s) d'entrée, appelée **enregistrement**, exécuter les *actions* dont la *condition* est remplie ou absente. Les *conditions* des *actions* sont testées selon leur ordre d'apparition dans le programme.

### Les actions

Une *action* est une suite d'instructions du langage, comprenant la création de variables (et tableaux associatifs) par affectation, des instructions conditionnelles et répétitives. Les opérateurs, les fonctions utilisateur et prédéfinies, les expressions régulières étendues, ainsi que des instructions d'entrée/sortie rendent AWK très riche et fonctionnel. Seules quelques possibilités sont décrites ici.

Le retour à la ligne et le point-virgule sont des séparateurs d'instruction. Notons que sur une ligne, tout ce qui suit le caractère **#** (non protégé) est un commentaire.

### Exemple

Le fichier *prog2.awk* de l'exemple précédent :

```
#!/usr/bin/awk -f

{
    print $1
}
```

ne contient que l'instruction **print \$1** qui n'est sujette à aucune condition. Elle est exécutée à chaque ligne lue. Pour **awk**, la ligne shebang est un commentaire.

□

9. On peut aussi définir des fonctions.

## L'instruction print

print est une instruction permettant d'écrire sur la sortie ou dans un fichier. Plusieurs emplois sont possibles :

- **print**  
écrit sur la sortie l'enregistrement courant **\$0** (voir plus loin). L'écriture est suivie du contenu de la variable **ORS** ;
- **print expression {, expression}**  
écrit sur la sortie chaque *expression* (résultat) en les séparant par le contenu de la variable **ORS**. L'écriture est suivie du contenu de la variable **ORS** ;
- **print expression {, expression} >[>] fichier**  
Comme précédemment mais l'écriture se fait dans le fichier *fichier*. S'il n'y a qu'un **>**, le fichier est écrasé. S'il est doublé, l'écriture est ajoutée en fin du fichier.

## Exemple

```
$ cat prog3.awk
#!/usr/bin/awk -f

# traitement pour toutes les lignes
{
    # écrit ce message pour chaque ligne
    print "un enregistrement lu..."

    # et l'opération qui somme le premier champ et 10
    print $1 " + 10 =", $1 + 10
}
$ cat fic3
10.5
28
-5.2
$ ./prog3.awk fic3
un enregistrement lu...
10.5 + 10 = 20.5
un enregistrement lu...
28 + 10 = 38
un enregistrement lu...
-5.2 + 10 = 4.8
```

□

## Les blocs BEGIN et END

Il existe deux conditions fictives : **BEGIN** et **END**. Le bloc **BEGIN** :

```
BEGIN {
    action
}
```


n'est exécuté qu'une seule fois, avant même l'ouverture du premier fichier d'entrée. Il sert principalement à initialiser un certain nombre de variables.

Le bloc **END** :

```
END {
    action
}
```

n'est exécuté qu'une seule fois, lorsque tous les fichiers et enregistrements ont été traités. Il peut servir à exploiter un résultat global.

Ainsi, seules les conditions différentes de **BEGIN** et **END** peuvent être satisfaites par les enregistrements des fichiers d'entrée.

 Par défaut, les enregistrements sont des lignes. On peut modifier la variable **RS** pour indiquer autre chose que le retour à la ligne comme délimiteur. Si l'on affecte à **RS** un seul caractère, c'est le délimiteur à utiliser. Si on lui affecte une chaîne, c'est une expression régulière étendue (voir plus loin).

### Exemple

```
$ cat prog4.awk
#!/usr/bin/awk -f

BEGIN {
    print "DEBUT >>>>>"
}

{
    print "premier champ de la ligne :", $1
}

END {
    print "FIN <<<<<<<"
}

$ cat fic1
1a 1b 1c
2a 2b 2c
3a 3b 3c
$ cat fic3
10.5
28
-5.2
$ ./prog4.awk fic1 fic3
DEBUT >>>>>
premier champ de la ligne : 1a
premier champ de la ligne : 2a
premier champ de la ligne : 3a
premier champ de la ligne : 10.5
premier champ de la ligne : 28
premier champ de la ligne : -5.2
FIN <<<<<<<
```

□

### Champs d'un enregistrement


Lors de leur lecture, les enregistrements sont découpés en **champs**. Le séparateur de champ est stocké dans la variable **FS**. Par défaut, c'est un espace, et les champs sont séparés par des suites d'espaces/tabulations. L'option

**-Fséparateur** de **awk** fixe la valeur de **FS** à *séparateur*. On peut modifier sa valeur dans n'importe quel bloc, notamment dans le bloc **BEGIN**.

Si **FS** ne contient qu'un caractère (autre qu'espace), c'est le délimiteur de champ. S'il contient la chaîne vide, tout caractère de l'enregistrement est un champ. Tout autre chaîne contenue dans **FS** est traitée comme une expression régulière étendue. Notons que la valeur espace pour **FS** est un cas particulier car dans les autres cas, plusieurs délimiteurs de champs consécutifs séparent des champs vides.


Les champs obtenus sont désignés par **\$1**, **\$2**, etc. L'enregistrement lui-même est désigné par **\$0**. Ces "variables" peuvent être utilisées dans les conditions et/ou les actions.

Notons que les variables (voir ci-dessous) offrent un moyen plus souple de désigner les enregistrements : si une variable **n** vaut 2, alors **\$n** désigne le 2<sup>e</sup> champ et **\$(n+1)** désigne le 3<sup>e</sup>.


 La variable **NF** indique le nombre de champs extraits de l'enregistrement courant.

### Exemples

```
$ cat prog5a.awk
{
    print "NF =", NF, " et champ1 =", $1
}
$ cat fic5
1a:1b:1c
2a::2b::2c
:3a:3b:3c
$ awk -f prog5a.awk fic5
NF = 1  et champ1 = 1a:1b:1c
NF = 1  et champ1 = 2a::2b::2c
NF = 1  et champ1 = :3a:3b:3c
```

 *il n'y a pas de blanc dans le fichier d'entrée donc les lignes ne comportent qu'un seul champ*

```
$ awk -F: -f prog5a.awk fic5
NF = 3  et champ1 = 1a
NF = 5  et champ1 = 2a
NF = 4  et champ1 =
```

 *mais en utilisant : comme séparateur, c'est différent. Notons que la troisième ligne commence par un champ vide*

```
$ cat prog5b.awk
BEGIN {
    FS=":"
}

{
    print "NF =", NF, " et champ1 =", $1
}
```

 *La variable **FS** est modifiée dans le bloc **BEGIN** et il n'est plus nécessaire d'utiliser l'option **-F***

```
$ awk -f prog5b.awk fic5
```

```
NF = 3  et champ1 = 1a
```

```
NF = 5  et champ1 = 2a
```

```
NF = 4  et champ1 =
```

□

## Variables utilisateur et chaînes

Les variables dans AWK ne sont pas typées. Elles peuvent contenir des valeurs entières, des flottants ou des chaînes. Selon le contexte, **awk** utilisera leur contenu comme une chaîne ou un nombre. Elles sont créées dynamiquement, dans n'importe quel bloc. Une variable non initialisée contient la chaîne vide ou 0 selon son emploi.

Pour créer/initialiser/modifier une variable, on utilise l'affectation. Le caractère marquant la décimale est le point. Une chaîne est une suite de caractères encadrée par des guillemets ("). La concaténation de chaînes se fait simplement en les faisant se suivre d'un espace. On obtient le contenu d'une variable simplement en utilisant son *identificateur*.

### Exemples (généraux)

```
var1=10
```

⇒ crée une variable en utilisant une valeur entière

```
var2 = 12.5
```

⇒ crée une variable en utilisant un flottant

```
var3 = "bonjour"
```

⇒ crée une variable en utilisant une chaîne

```
var4 = var3 ", vous !"
```

⇒ crée une variable en concaténant deux chaînes, dont la première est le contenu de **var3**

```
var5 = var1 var2
```

⇒ crée une variable en concaténant les deux chaînes "10" et "12.5"! Cette variable pourra être ensuite utilisée comme un nombre...

□

### Exemple (plus pratique)

```
$ cat prog6.awk
```

```
BEGIN {
```

```
    sum1=0
```

```
    sum2=0
```

```
}
```

```
{
```

```
    sum1+=$1
```

```
    sum2+=$2
```

```
}
```

```
END {
```

```
    print sum1 "\t" sum2
```

```
}
```

⇒ ce programme écrit la somme totale des premiers champs suivie de celle des deuxièmes champs des fichiers lus

```
$ cat fic6a
10.5      24.3
28        51.4
5         -2
$ cat fic6b
-3.5      18
243       11.1
21.2      100
$ awk -f prog6.awk fic6a fic6b
304.2     202.8
```

□

### Quelques variables prédéfinies

Outre les variables **RS**, **FS**, **NF**, **\$0**, **\$1**, etc., **awk** utilise/fournit d'autres variables, parmi lesquelles :

- **NR**  
le nombre total d'enregistrements lus jusqu'à présent
- **FNR**  
le numéro de l'enregistrement courant, dans le fichier courant
- **FILENAME**  
le nom du fichier d'entrée en cours de traitement. Contient – s'il s'agit de l'entrée standard
- **FIELDWIDTHS**  
(vide par défaut) une liste de largeurs de champs séparées par des espaces. Les champs seront découpés en utilisant les longueurs indiquées, sans tenir compte de **FS**
- **OFS**  
(espace par défaut) caractère à utiliser pour séparer les champs en sortie (voir **print**)
- **ORS**  
(retour à la ligne par défaut) caractère à utiliser pour séparer les enregistrements en sortie (voir **print**)

### Expression des conditions d'exécution

Outre **BEGIN** et **END**, les conditions d'exécution des actions peuvent être :

- */ expreg /*  
exécuter l'action si l'enregistrement courant vérifie l'expression régulière étendue *expreg* ;
- *expression relationnelle*  
exécuter l'action si l'enregistrement courant vérifie l'*expression relationnelle* ;
- *condition<sub>1</sub> && condition<sub>2</sub>*  
exécuter l'action si les deux conditions sont vérifiées ;
- *condition<sub>1</sub> || condition<sub>2</sub>*  
exécuter l'action si l'une des deux conditions est vérifiée ;
- *condition<sub>1</sub> ? condition<sub>2</sub> : condition<sub>3</sub>*  
si la *condition<sub>1</sub>* est vérifiée, exécuter l'action si la *condition<sub>2</sub>* l'est aussi. Si la *condition<sub>1</sub>* n'est pas vérifiée, exécuter l'action si la *condition<sub>3</sub>* l'est ;
- *(condition)*  
groupe une condition ;

- **!** *condition*  
exécuter l'action si la condition n'est pas vérifiée ;
- *condition<sub>1</sub>*, *condition<sub>2</sub>* décrit un intervalle. L'action est exécutée pour les enregistrements compris entre un qui vérifie *condition<sub>1</sub>* jusqu'à un vérifiant *condition<sub>2</sub>*, compris.

## Expressions régulières étendues

Le langage AWK utilise des expressions régulières étendues. La seule différence avec les expressions régulières simples est que les caractères **?**, **+**, **(**, **)** et **{** sont spéciaux pour les expressions régulières étendues (et ont la même signification quand ils sont précédés de **\** dans la version simple).

Notons que le point (.) peut remplacer aussi le retour à la ligne, notamment si on utilise une expression régulière pour **RS**.

## Opérateurs et expressions relationnelles

AWK reconnaît de nombreux opérateurs, parmi lesquels, dans l'ordre décroissant de priorité :

<b>(...)</b>	groupement
<b>\$</b>	référence de champ
<b>++ --</b>	pré/post incrémentation/décrémentation
<b>^</b>	puissance (diffère du C)
<b>+ - !</b>	plus unaire, moins unaire, négation
<b>* / %</b>	multiplication, division, modulo
<b>+ -</b>	addition, soustraction
<i>espace</i>	concaténation de chaînes
<b>&lt; &gt;</b>	opérateurs de comparaison qui
<b>&lt;= &gt;=</b>	s'appliquent aussi bien à des nombres qu'à
<b>! = ==</b>	des chaînes
<b>~ !~</b>	correspondance (non correspondance) avec une expression régulière : <i>exp ~ /expreg/</i> est vrai si l'expression <i>exp</i> vérifie l'expression régulière <i>expreg</i>
<b>&amp;&amp;</b>	et logique
<b>  </b>	ou logique
<b>? :</b>	l'opérateur conditionnel du C
<b>= += -=</b>	affectation simple et
<b>*= /= %= ^=</b>	affectations avec opérations

## Exemple

```
$ cat fic7a
num1      num2
-----
10.5      24.3
28        51.4
5         -2
$ cat fic7b
num1      num2
-----
-3.5      18
243       11.1
21.2      100
$ cat prog7.awk
BEGIN {
```



```

    sum1=0
    sum2=0
    print "num1\t num2\t somme"
    print "-----"
}

FNR > 2 {
    print $1 "\t" $2 "\t" ($1 + $2)
    sum1+=$1
    sum2+=$2
}

END {
    print "-----"
    print sum1 "\t" sum2 "\t" (sum1 + sum2)
}

```

⇒ *ce programme écrit les données des fichiers d'entrée (en ignorant les deux premières lignes de chaque fichier) en ajoutant une colonne `somme` contenant la somme des deux premières colonnes. Il calcule aussi la somme totale de chaque colonne et l'écrit en dernier.*

```
$ awk -f prog7.awk fic7a fic7b
```

num1	num2	somme
-----		
10.5	24.3	34.8
28	51.4	79.4
5	-2	3
-3.5	18	14.5
243	11.1	254.1
21.2	100	121.2
-----		
304.2	202.8	507

```
$ cat prog8.awk
```

```

BEGIN {
    sum1=0
    sum2=0
    fichier=""
}

fichier == "" {
    fichier=FILENAME
}

fichier != "" && fichier != FILENAME {
    print fichier " : " sum1 "\t" sum2
    fichier=FILENAME
    sum1=0
    sum2=0
}

FNR > 2 {
    sum1+=$1
    sum2+=$2
}

```

```
END {  
    print fichier " : " sum1 "\t" sum2  
}
```

⇒ *Un peu plus élaboré que le précédent, ce programme écrit, pour chaque fichier d'entrée, une ligne avec le nom du fichier ainsi que la somme pour chaque colonne. L'utilisation d'instructions conditionnelles aurait permis d'en alléger le code. La gestion de la variable `fichier` permet de détecter le passage à la lecture du prochain fichier, et afficher ainsi le résultat pour le fichier précédent.*

```
$ awk -f prog8.awk fic7a fic7b  
fic7a : 43.5      73.7  
fic7b : 260.7     129.1
```

□

# Chapitre 13

## Autres manipulations de fichiers

---

### 13.A Opérations diverses

#### 13.A.1 file : analyser le contenu d'un fichier

Les extensions des noms de fichiers étant libres, il n'est pas évident de savoir ce que contient un fichier sans risquer de faire une bêtise. La commande externe **file** analyse le contenu des fichiers (ou répertoires) passés en arguments et indique ce qu'il contient (lorsqu'elle arrive à le déterminer...).

#### Synopsis

**file** référence {référence}

#### Exemples

```
$ ls -ld public mbox algo.cxx /bin/bash /dev/sda1
-rwxr-xr-x    1 root    root      258288 mar  7  2000 /bin/bash
brw-rw----    1 root    disk        8,   1 mai  5  1998 /dev/sda1
drwx-----   5 cpb     prof      1024 sep 13 07:05 public
-rw-r--r--    1 cpb     prof      1063 sep 19 01:06 algo.cxx
-rw-----    1 cpb     prof        551 sep  5 11:35 mbox
$ file public mbox algo.cxx /dev/sda1 /bin/bash
public:      directory
mbox:        English text
algo.cxx:    ASCII C program text
/dev/sda1:   block special (8/1)
/bin/bash:   ELF 32-bit LSB executable, Intel 80386, version 1, dynamically
linked (uses shared libs), stripped
```

⇒ **file** indique que le premier fichier est un répertoire, le second est un fichier contenant du texte en anglais, le troisième est le source d'un programme C++, le quatrième est (une partition d')un disque dur. Le dernier fichier est un fichier binaire exécutable au format ELF (Executable and Linking Format).

### 13.A.2 split : découper un fichier en plusieurs fichiers

Lorsqu'on veut sauvegarder un fichier trop volumineux pour le support de destination (comme une disquette, une clé USB, etc.), il n'y a pas d'autre choix, après une compression, que de le découper en plusieurs fichiers qui peuvent « tenir » sur ce support. C'est l'utilisation la plus courante de la commande **split**.

#### Synopsis

```
split ( [-l n] | [-b n[b|k|m]] ) [-a taille_suffixe] [référence [préfixe_sortie]]
```

Sans option, **split** découpe le fichier *référence* (sans toutefois le modifier) en autant de fichiers qu'il en faut, contenant chacun (sauf le dernier) 1000 lignes. Si *référence* n'est pas spécifiée ou est le caractère **-**, **split** découpe ce qui est lu sur son entrée standard.

Les options **-l** et **-b** sont mutuellement exclusives :


- L'option **-l** *n* permet de spécifier le nombre de lignes (*n*) à copier dans les fichiers de sortie (elle est activée par défaut avec *n* = 1000) ;
- L'option **-b** *n* permet de spécifier la taille (*n*) maximale des fichiers de sortie. Par défaut, *n* est exprimé en octets mais on peut le faire suivre d'une unité : **b** pour blocs de 512 octets, **k** pour blocs d'1 Ko (1024 octets), et **m** pour blocs d'un Mo (1024<sup>2</sup> octets). Cette option est la plus utilisée car elle permet de découper un fichier volumineux pour le sauvegarder sur des supports dont on connaît la capacité (disquettes, clés USB, CD-R, etc.).

Le nom des fichiers obtenus par le découpage commence par *préfixe\_sortie* (qui vaut **x** par défaut), suivi d'un **suffixe**. Par défaut, le suffixe est constitué de deux caractères alphabétiques. L'option **-a** demande que le suffixe occupe *taille\_suffixe* caractères alphabétiques.

Le nom des fichiers créés est ordonné. Si on prend une taille de suffixe de 2 caractères (le défaut), alors :

- le premier fichier de sortie se nomme *préfixe\_sortieaa* (par défaut, *xaa*),
- le second se nomme *préfixe\_sortieab* (par défaut, *xab*),
- ...
- le 27<sup>e</sup> se nomme *préfixe\_sortieba*,
- etc. jusqu'au dernier fichier possible qui est *préfixe\_sortiezz* (par défaut *xzz*).

Par défaut il ne peut y avoir plus de 26<sup>2</sup> fichiers de sortie, sinon cela produit une erreur. En utilisant l'option **-a**, cette limite est de 26<sup>*taille\_suffixe*</sup>.

 En triant alphabétiquement les fichiers de sortie, on les obtient dans l'ordre dans lequel ils ont été créés. Un simple motif de noms de fichiers permet d'obtenir ces noms dans l'ordre<sup>1</sup>.

 Pour obtenir le fichier d'origine, il suffit de concaténer les fichiers de sortie.

#### Exemples (découpage par le nombre de lignes)

```
$ ls -l
total 4
-rw-r--r--  1 joe      motard      677 nov 26 14:58 cigale.txt
$ wc -l cigale.txt
21 cigale.txt
```

1. On rappelle qu'un motif de noms de fichiers est remplacé par la **liste triée** des noms de fichiers qui correspondent au motif.

⇒ *cigale.txt* est un fichier texte contenant 21 lignes

```
$ split -l 5 cigale.txt
```

⇒ découpage de *cigale.txt* en fichiers de 5 lignes

```
$ ls -l
```

```
total 24
```

```
-rw-r--r--  1 joe      motard      677 nov 26 14:58 cigale.txt
-rw-r--r--  1 joe      motard      109 nov 26 15:38 xaa
-rw-r--r--  1 joe      motard      234 nov 26 15:38 xab
-rw-r--r--  1 joe      motard      142 nov 26 15:38 xac
-rw-r--r--  1 joe      motard      163 nov 26 15:38 xad
-rw-r--r--  1 joe      motard       29 nov 26 15:38 xae
```

⇒ **split** a créé les fichiers *xaa* à *xae*. Ils sont de tailles différentes car les lignes ne contiennent pas toutes le même nombre de caractères.

```
$ cat xaa
```

```
Texte initial:
```

```
La cigale et la fourmi de Jean de La Fontaine.
```

```
La cigale ayant chante tout l'été,
```

⇒ *xaa* contient bien 5 lignes

```
$ cat xae
```

```
Eh bien! dansez maintenant."
```

⇒ alors que *xae* n'en contient qu'une

```
$ cat x* > cigale.new
```

⇒ on recrée le fichier d'origine simplement avec **cat**, les motifs de noms de fichiers et les redirections

```
$ ls -l cigale.*
```

```
-rw-r--r--  1 joe      motard      677 nov 26 15:38 cigale.new
-rw-r--r--  1 joe      motard      677 nov 26 14:58 cigale.txt
```

⇒ *cigale.txt* et le nouveau fichier (*cigale.new*) ont la même taille

```
$ diff cigale.*
```

⇒ ils sont bien identiques car la commande **diff** (vue dans l'annexe [B.2 page 291](#)) reste muette !

□

### Exemples (découpage par la taille)

```
$ ls -l
```

```
total 28876
```

```
-rw-r--r--  1 joe      motard  29528612 nov 26 15:28 linux-2.4.22.tar.bz2
```

⇒ *linux-2.4.22.tar.bz2* est un gros fichier binaire contenant les sources compressés du noyau Linux 2.4.22

```
$ split -b 1450000 linux-2.4.22.tar.bz2 partie-
```

⇒ découpage de *linux-2.4.22.tar.bz2* en fichiers de taille 1 450 000 octets maximum.

```
$ ls -l
total 57880
-rw-r--r-- 1 joe motard 29528612 nov 26 15:28 linux-2.4.22.tar.bz2
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-aa
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-ab
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-ac
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-ad
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-ae
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-af
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-ag
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-ah
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-ai
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-aj
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-ak
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-al
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-am
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-an
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-ao
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-ap
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-aq
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-ar
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-as
-rw-r--r-- 1 joe motard 1450000 nov 26 15:45 partie-at
-rw-r--r-- 1 joe motard 528612 nov 26 15:45 partie-au
```

⇒ chaque fichier obtenu commence par *partie-* et peut tenir sur une disquette...

□

### 13.A.3 curl : télécharger ou télédéposer une URL

La commande **curl** permet de transférer des données depuis/vers un serveur en utilisant l'un des nombreux protocoles supportés (HTTP, HTTPS, FTP, SCP, SFTP, TFTP, ...). De part le nombre de protocoles et de réglages permis, **curl** admet un nombre très important d'options. Nous nous limiterons à un usage très simplifié.

#### Synopsis

```
curl [-g] [--create-dirs] [--remote-name-all] {-o référence} {-O} [options-spécifiques] {url}
```

Les arguments optionnels *url* sont des URL (*Uniform Resource Locator*) dont la forme dépend du protocole utilisé. Par exemple :

```
http://curl.haxx.se
http://curl.haxx.se/docs/manual.html
ftp://ftp.rfc-editor.org/in-notes/rfc3986.txt
sftp://serveur.org/~parici/fichier
scp://serveur.org:20022/~parici/fichier
```

sont des URL valides et commencent par le protocole d'accès à la **ressource**. Notons qu'au cas où le serveur écoute sur un port non standard pour le protocole, on précise le *port* à utiliser en ajoutant *:port* à la suite de son adresse IP (ou de son nom), comme on le voit sur la dernière URL.

De plus, **curl** permet de simplifier l'écriture de multiples URL à l'aide des accolades et des crochets :

- les accolades forment une écriture factorisée à l'aide d'un ensemble, par exemple :

```
http://site.{un,deux,trois}.com
```

- les crochets forment une écriture factorisée à l'aide d'intervalles pour des suites alphanumériques, par exemple :

```
ftp://ftp.numericals.com/file[1-100].txt
ftp://ftp.numericals.com/file[001-100].txt
ftp://ftp.letters.com/file[a-z].txt
```

L'utilisation de pas pour les intervalles est aussi admise :

```
http://www.numericals.com/file[1-100:10].txt
http://www.letters.com/file[a-z:2].txt
```

Les combinaisons sont admises mais pas leur imbrication. Au cas où les accolades ou les crochets doivent faire partie de l'URL, l'option **-g** désactive leur développement.

 N'oublions pas que les accolades et les crochets sont des caractères spéciaux pour bash...

Sans option, **curl** utilise le protocole indiqué par chaque *url* pour **récupérer la ressource ciblée et l'écrit sur sa sortie standard**. Cela permet à **curl** de figurer dans un *pipeline*.

### Exemple

```
$ curl http://infodoc.iut.univ-aix.fr
<html>

<head>
<meta http-equiv="Content-Language" content="fr">
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>DEPARTEMENT INFORMATIQUE&nbsp;&nbsp; IUT Aix</title>

<!--mstheme--><link rel="stylesheet" href="caps1001.css">

<meta name="Microsoft Theme" content="capsules 1001">

</head>

<body>
...

</body>

</html>
```

 *affichage à l'écran du fichier `index.html` du site `infodoc.iut.univ-aix.fr`.*

□

L'option **-o** *référence* écrit la sortie dans le fichier *référence* à la place de la sortie standard. Cette option peut s'utiliser autant de fois qu'il y a d'*url* spécifiées. De plus, en cas d'utilisation d'accolades ou de crochets, *référence* peut contenir le caractère **#** suivi d'un nombre *n*, ce qui sera remplacé par la portion de l'URL développée pour le *n*<sup>e</sup> groupement d'accolades/crochets.

L'option **--create-dirs**, utilisée conjointement avec l'option **-o**, demande à **curl** de créer les répertoires présents dans *référence* s'ils n'existent pas déjà.

L'option **-O** demande d'écrire la sortie dans un fichier de même nom que celui récupéré. Elle peut s'utiliser autant de fois qu'il y a d'*url* spécifiées. L'option **--remote-name-all** active l'option **-O** pour toutes les *url* qui suivent.

## Exemples

```
$ curl -o fic http://infodoc.iut.univ-aix.fr
```

% Total	% Received	% Xferd	Average Speed	Time	Time	Time	Current
			Dload Upload	Total	Spent	Left	Speed
100	1743	100	1743	0	0	213k	0
--:--:--	--:--:--	--:--:--	340k				

⇒ crée un fichier *fic* contenant la page *http://infodoc.iut.univ-aix.fr*. Remarquons que puisque la sortie n'est pas utilisée pour écrire la ressource, *curl* affiche (sur la sortie d'erreur) une information de progression

```
$ curl -o fic1 -o fic2 http://infodoc.iut.univ-aix.fr http://infodoc.iut.univ-aix.fr/~cpb
```

% Total	% Received	% Xferd	Average Speed	Time	Time	Time	Current
			Dload Upload	Total	Spent	Left	Speed
100	1743	100	1743	0	0	183k	0
--:--:--	--:--:--	--:--:--	340k				
100	428	100	428	0	0	135k	0
--:--:--	--:--:--	--:--:--	135k				

⇒ crée le fichier *fic1* contenant la page de la première URL et *fic2* contenant celle de la deuxième

```
$ curl --create-dirs -o ici/fic http://infodoc.iut.univ-aix.fr
```

...

⇒ crée le fichier *ici/fic* contenant la page, en créant si besoin le répertoire *ici*

```
$ curl -O http://infodoc.iut.univ-aix.fr/index.html
```

...

⇒ télécharge le fichier *index.html* en le sauvant sous le même nom. Notons que le nom du fichier est obligatoire dans l'URL

□

L'option **-u** *utilisateur[:motdepasse]* permet d'accéder à une ressource protégée par un nom d'utilisateur et un mot de passe. Si *motdepasse* n'est pas indiqué, il sera demandé si besoin par la suite. Notons que pour SSH et SSL, l'authentification par paire de clés est possible en utilisant les options **--key**, **--pubkey** et **--pass**.

L'option **-T** *référence* demande de déposer (*upload*) le fichier local *référence* sur à l'*url* spécifiée. Le nom que doit porter le fichier sur le serveur distant peut être spécifié à la fin de l'*url*. Si *référence* est **-**, **curl** transférera ce qu'elle lit sur l'entrée standard. À l'instar des *urls*, *référence* peut désigner plusieurs fichiers, si elle contient des accolades et/ou des crochets qui sont développés par **curl**. L'option **-T** peut s'utiliser autant de fois qu'il y a d'*url* spécifiées.

❗ Dans le contexte du téléchargement de pages Web, l'outil **wget** s'avère aussi très utile, car il est capable de télécharger un site complet. Pour cela, **wget** analyse les pages téléchargées et peut télécharger les liens qu'elles contiennent, ce que ne fait pas **curl**.

## 13.B Recherche de fichiers

### 13.B.1 find : rechercher des fichiers

La recherche de fichiers (ou répertoires) est réalisée par la commande externe **find**. Elle admet un grand nombre de critères possibles que l'on peut combiner en utilisant des opérateurs logiques. Elle permet même de fabriquer nos propres critères.

#### Synopsis

```
find {réf.rép} [expression]
```



*réf\_rép* sont des références à des répertoires existants. La recherche de fichiers correspondant aux critères mentionnés par *expression* va être opérée dans toute l'arborescence de chacun de ces répertoires (eux y compris). Tous les fichiers (cachés ou non) accessibles dans ces arborescences vont être passés, l'un après l'autre, au crible des critères exprimés dans *expression*.

*expression* est composée de **tests**, d'**options**, d'**actions** et d'**opérateurs logiques** permettant d'écrire des expressions complexes. Pour simplifier, on appellera *critère* les composants de *expression*. Un critère renvoie toujours une valeur, vraie ou fausse. L'expression est toujours évaluée de gauche à droite. Son évaluation s'arrête (pour le fichier examiné) dès que sa valeur est déterminée (*vraie* ou *fausse*). Puis, **find** passe au fichier suivant.

Si *expression* n'est pas présente, alors l'action **-print** est prise par défaut (et **find** affiche une référence de chaque fichier rencontré).

Le premier argument commençant par un des caractères `'-'`, `'('`, `','`, `':'` ou `'/'` marque le début de *expression*. Ce qui précède est considéré comme des *réf\_rép*. Si aucune *réf\_rép* n'est spécifiée, le répertoire de travail est pris par défaut.

### Opérateurs logiques :

Les opérateurs logiques servent à écrire des expressions complexes. Ils sont listés ci-dessous par ordre de priorité décroissante :

( *expression* )

(*groupement*) permet de grouper un ensemble de critères. Utile si précédé de **!** ou en présence de **-o**. **Attention**, les parenthèses sont des caractères spéciaux pour bash, il faut donc les protéger.

**!** *critère*

(*négation*) vrai ssi *critère* renvoie faux ;

**!** ( *expression* )

(*négation*) vrai ssi *expression* est fausse ;

*expression*<sub>1</sub> *expression*<sub>2</sub>

(*et*) renvoie faux sans évaluer *expression*<sub>2</sub> si *expression*<sub>1</sub> est fausse, sinon évalue *expression*<sub>2</sub> et renvoie sa valeur ;

*expression*<sub>1</sub> **-a** *expression*<sub>2</sub>

(*et*) comme *expression*<sub>1</sub> *expression*<sub>2</sub> ;

*expression*<sub>1</sub> **-o** *expression*<sub>2</sub>

(*ou*) renvoie vrai sans évaluer *expression*<sub>2</sub> si *expression*<sub>1</sub> est vraie, sinon évalue *expression*<sub>2</sub> et renvoie sa valeur.



**Tous les termes des expressions – y compris ( , ), !, -o et -a, ainsi que les différents critères – doivent être séparés par un ou plusieurs blancs.**

### Extrait des options reconnues :

La plupart des options figurant dans *expression* ont une portée globale, c'est à dire qu'elles s'appliquent avant même de commencer l'examen des fichiers. C'est le cas des options ci-dessous :

**-depth**

traiter d'abord l'arborescence d'un répertoire avant de traiter le répertoire lui-même. Par défaut, un répertoire est traité avant de traiter les fichiers et sous-répertoires qu'il contient. Cette option est automatiquement activée en présence de l'action **-delete** ;

**-maxdepth** *n*

avec  $n \geq 0$ , demande de ne pas descendre à une profondeur supérieure à *n* dans l'arborescence. Si *n* vaut 0, **find** ne descendra pas dans l'arborescence ;

**-mindepth** *n*

avec  $n \geq 0$ , demande de ne pas traiter les fichiers situés à une profondeur inférieure à *n*. Si *n* vaut 1, **find** ne traitera pas les *réf.rép* mais uniquement leur arborescence.

**Extrait des tests reconnus :****-type** *type*

vrai si et seulement si le fichier en cours d'analyse est du type *type*, où *type* peut être :

- f** pour un fichier ordinaire
- d** pour un répertoire
- l** pour un lien symbolique
- c** pour fichier spécial en mode caractère
- b** pour fichier spécial en mode bloc
- p** pour un tube nommé (fichier spécial)
- s** pour socket (fichier spécial)

**-empty**

vrai ssi le fichier est vide et est de type ordinaire ou répertoire ;

**-user** *nom*

vrai ssi *nom* est le propriétaire du fichier en cours d'analyse ;

**-uid** *num*

vrai ssi *num* est le numéro d'utilisateur du propriétaire du fichier en cours d'analyse ;

**-group** *groupe*

vrai ssi *groupe* est le groupe du fichier ;

**-gid** *num*

vrai ssi *num* est le numéro de groupe du fichier ;

**-name** *motif*

vrai ssi le nom du fichier (et non pas sa référence) correspond au *motif*. La description du *motif* est la même que pour *bash*, sans les accolades et où les **\***, **?** et **[]** peuvent correspondre au point qui commence le nom d'un fichier caché. Mais il faut protéger les caractères spéciaux pour ne pas qu'ils soient interprétés par le shell ;

**-iname** *motif*

comme **-name** mais sans distinguer les minuscules des majuscules ;

**-newer** *référence*

vrai ssi le fichier est plus récent que le fichier *référence* ;

**-mtime** [**+** | **-**] *n*

vrai ssi la dernière modification du fichier remonte à plus de (**+**), moins de (**-**), ou exactement *n* jours. Attention, le | entre le **+** et le **-** marque une alternative entre ces deux caractères, il n'est pas à écrire ;

**-atime** [**+** | **-**] *n*

vrai ssi le dernier accès au fichier a eu lieu il y a plus de (**+**), moins de (**-**), ou exactement *n* jours ;

**-perm** [**+** | **-**] *mode*

vrai ssi les permissions du fichier correspondent exactement au *mode* spécifié (de façon absolue ou symbolique). Si le *mode* est précédé de **-**, les permissions doivent être au moins celles indiquées. Si le mode est précédé de **+**, il faut juste qu'une permission du fichier corresponde à au moins une des permissions indiquées, que ce soit pour le propriétaire, le groupe ou les autres. Par exemple, **-perm +777** demande que le fichier ait au moins une permission, quelle que soit la catégorie d'utilisateur (la permission **---r-----** suffit) ;

**-size** [**+** | **-**] *n* [**b** | **c** | **k** | **M** | **G**]

vrai ssi le fichier a une taille à plus de (**+**), moins de (**-**), ou exactement *n* blocs de 512 octets (**b**, étant le défaut), octets (**c**), kilo-octets (**k**), méga-octets (**M**), giga-octets (**G**) ;

**Extrait des actions reconnues :****-print**

renvoie toujours vrai. Provoque l'affichage de la référence du fichier en cours d'analyse. Cette référence commence par *réf.rép/*. Si cette action n'est pas présente, elle est automatiquement ajoutée en dernier, sauf si les actions **-exec** ou **-ok** ont été exécutées ;

**-exec** *commande* ;

vrai ssi *commande* retourne une valeur vraie<sup>2</sup>. *commande* est donc exécutée. Elle peut comporter des options et des arguments, mais pas de redirections. Le point virgule indique où *commande* se termine. Il doit être protégé car c'est un caractère spécial du shell. Dans *commande*, on peut utiliser l'expression { } qui sera remplacée par la référence du fichier en cours d'analyse ;

**-ok** *commande* ;

même chose que **-exec** mais un message demande s'il faut exécuter la commande (réponse **y** ou **Y**) ou pas (dans ce cas **-ok** renvoie faux) ;

**-delete**

efface le fichier en cours d'analyse. Renvoie vrai si cela réussit. L'échec de la suppression provoque l'affichage d'un message d'erreur.

**Exemples**

La commande suivante recherche dans l'arborescence de */var* les fichiers ordinaires dont le nom est *cpb*, et affiche leur référence :

```
$ find /var -type f -name cpb -print
find: /var/lib/nfs/sm: Permission non accordée
find: /var/lib/nfs/sm.bak: Permission non accordée
find: /var/lib/pgsql: Permission non accordée
find: /var/lib/slocate: Permission non accordée
find: /var/log/samba: Permission non accordée
find: /var/log/squid: Permission non accordée
/var/spool/mail/cpb
find: /var/spool/at: Permission non accordée
find: /var/spool/cron: Permission non accordée
find: /var/spool/news/.xauth: Permission non accordée
find: /var/spool/squid: Permission non accordée
find: /var/spool/intercheck/infected: Permission non accordée
find: /var/gdm: Permission non accordée
```

☞ toutes les lignes commençant par **find:** sont des messages d'erreur. Seule la ligne */var/spool/mail/cpb* indique un fichier correspondant aux critères.

2. La valeur de retour d'une commande sera vue section 16.B.1 page 237

Dans la commande précédente, l'action **-print** figure à la fin pour que l'affichage de la référence des fichiers ne soit fait que s'ils vérifient les tests **-type** et **-name**. Si **-print** se trouve ailleurs, cela change considérablement l'affichage, comme le montre l'exemple ci-dessous (seules les premières lignes du résultat sont reportées ici) :

```
$ find /var -print -type f -name cpb
/var
/var/tmp
/var/tmp/A8_6Cadre.java.swp
/var/tmp/DSCF0054.JPG
/var/tmp/DSCF0055.JPG
/var/tmp/DSCF0057.JPG
/var/tmp/tp4.txt.swp
/var/tmp/DemineurTheme.java.swp
/var/lib
/var/lib/rpm
/var/lib/rpm/packages.rpm
/var/lib/rpm/nameindex.rpm
/var/lib/rpm/fileindex.rpm
/var/lib/rpm/providesindex.rpm
/var/lib/rpm/requiredby.rpm
/var/lib/rpm/conflictsindex.rpm
... .. etc.
```

La commande suivante recherche dans l'arborescence du répertoire de travail de l'utilisateur, les fichiers ordinaires dont l'extension est *adb*, *ads* ou *squ*, et affiche leur référence :

```
$ find -type f \( -name '*.ad[bs]' -o -name '*.squ' \) -print
./tpada/p_rationnels_g.ads
./tpada/p_rationnels_g-io.adb
./tpada/p_rationnels_g-io.ads
./tpada/tscl_rationnels.adb
./tpada/ts_rationnels.adb.squ
```

Mieux encore, en remplaçant **-print** par **-ok**, **find** peut demander s'il faut imprimer chaque fichier trouvé sur l'imprimante de la salle A. Le critère **-exec** est alors utilisé pour afficher que l'impression du fichier est lancée. La ligne de commandes devenant grande, elle est présentée sur deux lignes :

```
$ find -type f \( -name '*.ad[bs]' -o -name '*.squ' \) -ok lpr -PA {} \;
-exec echo "Impression envoyée" \;
< lpr ... ./tpada/p_rationnels_g.ads > ? y
Impression envoyée
< lpr ... ./tpada/p_rationnels_g-io.adb > ? n
< lpr ... ./tpada/p_rationnels_g-io.ads > ? n
< lpr ... ./tpada/tscl_rationnels.adb > ? y
Impression envoyée
< lpr ... ./tpada/ts_rationnels.adb.squ > ? n
```

➡ selon la réponse donnée à **find**, celle-ci exécute ou non la commande du critère **-ok** (puis celle de **-exec**). Notons que dans cet exemple, le code de retour de **lpr** et de **echo** est toujours vrai.

□

### 13.B.2 locate : localiser rapidement un fichier par son nom

La commande **locate** effectue une recherche rapide de fichiers à partir de leur nom.

#### Synopsis

**locate** [-bei] [--regex] *nom* { *nom* }

**locate** utilise une (ou plusieurs) base de données préparée par **updatedb** (voir section 13.B.3 page suivante) et affiche les références des fichiers de la base, qui correspondent à un *nom*, à raison d'une référence par ligne. La recherche est rapide car elle est menée uniquement sur la base et non sur le système de fichiers. En contrepartie, le résultat peut être obsolète si un fichier a été créé/déplacé/renommé/supprimé après la dernière mise à jour de la base. Toutefois, l'option **-e** demande à vérifier que la référence existe toujours avant de l'afficher.

Chaque *nom* est traité comme un motif de noms de fichiers. S'il ne contient pas de caractères de motifs, il est traité comme *\*nom\**, à moins qu'il ne contienne un \. Par défaut, une référence correspond à *nom* si sa totalité correspond à *nom*. L'option **-i** demande à ne pas tenir compte de la casse. L'option **-b** demande à ne comparer que le nom du fichier (*basename*) et non pas toute sa référence. L'option **--regex** demande à traiter les *noms* comme des expressions régulières.

### Exemples

```
$ locate xnest
```

```
/usr/share/doc/xnest
/usr/share/doc/xnest/changelog.Debian.gz
/usr/share/doc/xnest/changelog.gz
/usr/share/doc/xnest/copyright
/usr/share/gdm/applications/gdmflexiserver-xnest.desktop
/usr/share/icons/hicolor/16x16/apps/gdm-xnest.png
/usr/share/icons/hicolor/32x32/apps/gdm-xnest.png
/usr/share/icons/hicolor/scalable/apps/gdm-xnest.svg
/usr/share/pixmaps/gdm-xnest.png
/var/lib/dpkg/info/xnest.list
/var/lib/dpkg/info/xnest.md5sums
```

⇨ donne le même résultat que **locate \*xnest\***, c'est à dire les références qui contiennent *xnest*

```
$ locate 'xnest'
```

```
/usr/share/doc/xnest
```

⇨ réduit la recherche aux fichiers se terminant par *xnest*

```
$ locate -i 'xnest'
```

```
/usr/bin/Xnest
/usr/share/doc/xnest
```

⇨ de même mais sans tenir compte de la casse

```
$ locate '\xnest'
```

```
$ locate -i '\xnest'
```

⇨ aucune référence de la base est uniquement *xnest*

```
$ locate -b 'xnest'
```

```
/usr/share/doc/xnest
/usr/share/gdm/applications/gdmflexiserver-xnest.desktop
/usr/share/icons/hicolor/16x16/apps/gdm-xnest.png
/usr/share/icons/hicolor/32x32/apps/gdm-xnest.png
/usr/share/icons/hicolor/scalable/apps/gdm-xnest.svg
/usr/share/pixmaps/gdm-xnest.png
/var/lib/dpkg/info/xnest.list
/var/lib/dpkg/info/xnest.md5sums
```

⇨ (identique à **locate -b '\*xnest\*'**) seul le nom du fichier est comparé à *\*xnest\**

```
$ locate -b '\xnest'
/usr/share/doc/xnest
```

⇒ recherche un fichier dont le nom est exactement *xnest*.

□

### 13.B.3 updatedb : créer ou mettre à jour une base pour locate

La commande **updatedb** crée ou met à jour une base utilisable par **locate**.

#### Synopsis

**updatedb**

**updatedb** tient compte de son fichier de configuration `/etc/updatedb.conf` afin de créer ou mettre à jour la base `/var/lib/mlocate/mlocatedb.conf`. En principe, **updatedb** est exécutée quotidiennement par l'utilitaire **cron**. Par défaut, la base est construite en tenant compte de l'ensemble des fichiers du système, modulo des exclusions définies dans `/etc/updatedb.conf`. La base est destinée à être consultable par n'importe quel utilisateur au moyen de la commande **locate**. Cependant, **locate** n'affiche pas les références des fichiers auxquels l'utilisateur qui l'invoque n'a pas accès. Si besoin, root peut lever cette restriction.

Bien qu'elles ne soient pas indiquées ici, **updatedb** admet des options. Certaines permettant d'exclure des systèmes de fichiers, des noms de fichiers ou des chemins, pendant la construction de la base. Des exclusions sont déjà définies dans le fichier `/etc/updatedb.conf`. Il est aussi possible d'indiquer autre chose que `/` comme point de départ d'analyse pour la construction de la base. Enfin, une option permet à tout utilisateur d'exécuter **updatedb** pour créer une base personnelle limitée.

## 13.C Compression

Les algorithmes de compression de fichiers sont nombreux et n'ont pas été développés en même temps. C'est pourquoi, il existe plusieurs utilitaires de compression/décompression sous Unix parmi lesquels : **pack**, **compact**, **compress**, **gzip**, et **bzip2**.

**pack** et **compact** font partie de la préhistoire et ne sont plus utilisés aujourd'hui et ne seront donc pas détaillés dans ce qui suit.

### 13.C.1 compress/uncompress : la compression historique mais obsolète d'Unix

#### 13.C.1.a compress : compresser un fichier

L'utilitaire **compress** est assez ancien. Il n'accompagne plus les distributions récentes de Linux car il est moins efficace que les utilitaires **gzip** et **bzip2** (vus plus loin), et un peu parce que l'algorithme qu'il utilise était sujet à un copyright (qui touche aussi la compression utilisée pour le format d'image GIF). Cependant, on le rencontre encore souvent sur certains systèmes Unix.

#### Synopsis

**compress** [-vcr] {référence}

**compress**, tout comme les autres utilitaires de compression, ne compresse que des fichiers ordinaires. Sans option, **compress** remplace chaque fichier *référence* par un fichier portant le même nom mais avec l'extension **.z**. Si aucune *référence* n'est indiquée, la compression est opérée sur ce qui est lu sur l'entrée standard et son résultat est affiché sur la sortie standard. L'option **-c** demande à ce que les fichiers *référence* ne soient pas remplacés mais que la compression soit affichée sur la sortie standard. L'option **-r** (*recursive*) demande à ce que la compression soit effectuée récursivement sur les répertoires passés en arguments. L'option **-v** (*verbose*) demande que le taux de compression soit écrit sur la sortie d'erreur.

Notons que **compress** a un inconvénient majeur : il ne rajoute aucune information (somme de contrôle, par exemple) qui permettrait à un autre utilitaire (comme **uncompress**) de vérifier que le fichier compressé est bien intègre (entier et sans altération).

### 13.C.1.b uncompress : décompresser un fichier d'extension **.Z**

#### Synopsis

```
uncompress [-vc] { référence }
```

Chaque *référence* doit correspondre à un fichier compressé avec **compress**, d'extension **.z**. Il est remplacé par le fichier décompressé, où l'extension **.z** est supprimée. Sans argument, **uncompress** décompresse ce qui est lu sur son entrée standard (et devant correspondre à une compression opérée par **compress**). L'option **-c** demande à ce que *référence* ne soit pas remplacée mais que le résultat de sa décompression soit écrit sur la sortie standard. L'option **-v** demande à ce que le taux de la compression qui avait été opérée soit affiché sur la sortie d'erreur.

## 13.C.2 gzip/gunzip : la compression la plus répandue

### 13.C.2.a gzip : compression avec somme de contrôle

L'utilitaire de compression **gzip** est l'un des plus utilisés actuellement dans le monde Unix. Il a de nombreux avantages, notamment celui de rajouter une somme de contrôle dans le fichier compressé permettant de savoir si ce dernier n'a pas été altéré. Notons que la compression opérée par **gzip** suit l'algorithme dit de Lempel-Ziv.

#### Synopsis

```
gzip [-cdfINrv] [-chiffre] { référence }
```

Chaque fichier *référence* sera remplacé par son équivalent compressé ayant le même nom mais avec l'extension **.gz**. Si un fichier de ce nom existe déjà il ne sera pas écrasé à moins que l'option **-f** (*force*) soit spécifiée. En l'absence de *référence* **gzip** compressera ce qui est lu sur l'entrée standard mais uniquement si sa sortie n'est pas l'écran (c'est pas beau du binaire sur un terminal !!). Les options **-v** et **-c** ont la même signification que pour **compress**. L'option **-d** demande de décompresser au lieu de compresser, ce qui revient au même que la commande **gunzip** (voir ci-après). L'option **-l** demande d'afficher des informations sur les fichiers compressés passés en arguments. L'option **-r** demande d'effectuer la compression récursivement pour les répertoires passés en arguments. **-N** est une option activée par défaut et demande à ce que le nom du fichier à compresser ainsi que sa date de dernière mise à jour soient stockés dans le fichier compressé afin d'être restitués ultérieurement. Elle n'est utile qu'avec l'option **-l**. Enfin, l'option **-chiffre**, où  $1 \leq \text{chiffre} \leq 9$ , spécifie la vitesse de compression : 9 donnera la compression la plus lente mais la meilleure. Par défaut, *chiffre* vaut 6.



Les options **-v** et **-N** peuvent s'ajouter à l'option **-l** pour donner plus d'informations sur les fichiers.



### 13.C.2.b gunzip : effectuer différentes décompressions

Comme son nom l'indique, **gunzip** décompresse les fichiers compressés par **gzip**. En outre, **gunzip** (ou **gzip -d**) sait aussi décompresser un fichier compressé par **pack**, **compress** ou **zip** (s'il n'y a qu'un seul fichier dans l'archive).

#### Synopsis

```
gunzip [-cflNvr] {référence}
```

Chaque fichier *référence* doit être un fichier d'extension `.gz`, `.Z`, `.z`, `.tgz`, `.taz` ou un répertoire si la décompression récursive est demandée avec l'option **-r**. Chaque fichier *référence* sera décompressé par l'algorithme approprié et sera remplacé par un fichier de même nom sans l'extension.

**i** Les archives créées avec l'utilitaire **tar** (vu plus loin) et compressées par **gzip** portent souvent l'extension `.tgz`. Celles compressées par **compress** portent souvent l'extension `.taz`. Leur décompression donne un fichier d'extension `.tar`.

Les options **-c**, **-f**, **-v** et **-l** ont la même signification que pour **gzip**<sup>3</sup>. Pour les fichiers compressés par **gzip**, l'option **-N** demande à ce que le fichier décompressé porte le même nom et ait la même date de dernière modification que le fichier qui avait été compressé à l'origine.

### 13.C.2.c Les commandes en « z »

De nombreuses commandes se rapportant aux fichiers compressés existent. Les deux commandes suivantes opèrent une transformation du ou des fichiers passés en arguments :

- **gzexe** : destiné aux fichiers exécutables. Remplace les fichiers passés en arguments par des fichiers auto-extractibles. Un fichier auto-extractible est en fait un script capable d'extraire de son propre contenu le code exécutable compressé. Ce code est alors décompressé dans le répertoire `/tmp` puis exécuté ;
- **znew** : transforme des fichiers d'extension `.Z` (compressés par **compress**) en fichiers d'extension `.gz` (compression par **gzip**).

Les commandes suivantes ne modifient pas les fichiers passés en arguments. Elles effectuent un traitement sur ceux-ci comme s'il s'agissait de fichiers non compressés. Pour cela, elles décompressent parfois temporairement les fichiers indiqués (en plaçant les fichiers décompressés dans le répertoire `/tmp`) puis lancent la commande appropriée :

- **zcat** : équivalent à **gunzip -c** (et à **gzip -cd**), c'est la commande **cat** des fichiers compressés. Son rôle est donc d'afficher le contenu d'un fichier (dé)compressé, comme s'il n'avait jamais été compressé ;
- **zmore** : affiche le contenu de fichiers compressés, en paginant avec **more**
- **zless** : affiche le contenu de fichiers compressés, en paginant avec **less**
- **zgrep** : rechercher des lignes dans un fichier compressé, en utilisant **grep**
- **zcmp** : compare des fichiers compressés, en utilisant **cmp**
- **zdiff** : compare des fichiers compressés, en utilisant **diff**

3. On sait déjà que **gzip**, **gunzip** et **zcat** sont souvent un seul et même fichier qui adapte son comportement selon le nom par lequel il est appelé.



### 13.C.3 bzip2/bunzip2 : la compression la plus récente

#### 13.C.3.a bzip2 : plus d'efficacité et de robustesse dans la compression

**bzip2** est l'un des compacteurs les plus récents. Il est notamment utilisé par Linus Torvald pour diffuser les sources du noyau Linux. C'est le plus efficace de ceux mentionnés jusqu'ici. Il utilise l'algorithme de réduction par tri de bloc de Burrows-Wheeler, et le codage d'Huffman.

##### Synopsis

```
bzip2 [-cfdktv] [-chiffre] {référence}
```

Le comportement de **bzip2** est similaire à celui de **gzip** mais n'admet pas tout à fait les mêmes options. Les options **-c**, **-d**, **-f**, **-v** et **-chiffre** ont la même signification. En revanche, **bzip2** n'effectue pas de compression récursive.

Chaque fichier *référence* compressé par **bzip2** est remplacé par un fichier portant le même nom, et d'extension **.bz2**. L'option **-k** (*keep*) demande à garder les fichiers d'origine.

L'option **-t** (*test*) demande de vérifier que les fichiers passés en arguments sont des fichiers "bzip2" corrects. Elle reste muette et renvoie un code de retour à 0 si c'est le cas.

❶ **bzip2** compresse le(s) fichier(s) d'origine en créant des blocs de données compressées (contenant 900 000 octets par défaut) tous indépendants. Cela permet notamment de concaténer des fichiers "bzip2", mais aussi de restaurer les blocs corrects des fichiers "bzip2" altérés (voir **bzip2recover** plus loin).

#### 13.C.3.b bunzip2 : décompresser les fichiers bzip2

**bunzip2** est l'utilitaire de décompression des fichiers compressés par **bzip2**. Son utilisation est similaire à celle de **gunzip**.

❶ À l'instar de **gunzip**, **bunzip2** est identique à **bzip2 -d**, car c'est le même exécutable que **bzip2**. Il adapte son comportement selon le nom par lequel il est appelé.

##### Synopsis

```
bunzip2 [-ckfv] {référence}
```

où chaque *référence* doit être un fichier compressé par **bzip2**. Si l'extension de *référence* est **.bz2** ou **.bz**, il sera remplacé par un fichier de même nom sans cette extension. Si elle est **.tbz2** ou **.tbz**, il sera remplacé par un fichier d'extension **.tar**. Dans les autres cas, *référence* sera remplacé par le fichier *référence.out*.

Les options **-c**, **-f**, **-k** et **-v** ont la même signification que pour **bzip2**.

### 13.C.3.c Les commandes en « bz »

À l'instar de **gzip**, il existe des utilitaires travaillant directement sur les fichiers compressés par **bzip2** ou liés à cette compression, parmi lesquels :

- **bzcat**, qui n'est autre que **bzip2** et est équivalent à **bzip2 -cd** (et à **bunzip2 -c**) ;
- **bzgrep** qui recherche des lignes dans un fichier compressé ;
- **bzless** qui affiche en paginant le contenu d'un fichier compressé ;
- **bzme**, semblable à **znew**, permet de transformer des fichiers "gzip" (.gz), "compress" (.Z ou .z), en fichiers "bzip2" (.bz2). **bzme** est même capable de transformer des archives "zip" (.zip) ou des archives "tar" compressées avec gzip (.tgz), en fichiers .tar.bz2 ;
- **bzip2recover** est un utilitaire permettant de récupérer des parties de fichiers "bzip2" endommagés. En fait, **bzip2recover** extrait du fichier "bzip2" les blocs qu'il contient et crée un fichier pour chacun d'eux. Il suffit d'utiliser ensuite **bzip2 -t** pour vérifier ces fichiers et repérer ceux corrects, qui peuvent être décompressés indépendamment ou concaténés pour former un fichier "bzip2" correct.

## 13.D Archivage

### 13.D.1 tar : l'archiviste historique et incontournable d'Unix

À l'origine, **tar** (*tape archiver*) était destiné à la sauvegarde/restauration d'arborescences sur/depuis une bande magnétique. Aujourd'hui, **tar** est très utilisé pour regrouper plusieurs fichiers ou répertoires (arborescences) dans un seul fichier binaire appelé une « **archive** ». Le plus souvent les archives sont créées afin d'être facilement téléchargeables depuis l'Internet par FTP (*File Transfer Protocol*) qui ne permet pas de télécharger une arborescence en une seule commande<sup>4</sup>.

Une archive est un fichier particulier qui porte par convention l'extension .tar. Lorsqu'elle est compressée en même temps qu'elle est créée, elle porte l'extension .tgz pour la compression **gzip**, et l'extension .taz ou .tgZ si elle est compressée par **compress**. Souvent l'archive est compressée après avoir été créée. Elle porte alors l'extension .tar.gz pour **gzip**, .tar.z ou .tar.Z pour **compress**, et .tar.bz2 pour **bzip2**.

❗ Sous Windows, les archives sont le plus souvent créées (avec compression automatique) en utilisant le logiciel **WinZIP** et portent l'extension .zip. Notons que **WinZIP** sait extraire les fichiers contenus dans une archive .tar. En revanche, **tar** ne sait pas lire les archives « zip » : il faut utiliser l'utilitaire **zip** (vu plus loin).

### Synopsis

```
tar [-] action{ option} { référence}
```

L'utilisation du - devant l'*action* est optionnelle mais conseillée (sauf pour des versions BSD où elle peut être interdite). Il faut obligatoirement spécifier **une action et une seule** avec l'une des lettres **c**, **x**, **t**, **r** ou **u**, commandant :

- **c** : (*create*) la création d'une archive ;
- **x** : (*extract*) l'extraction du contenu d'une archive ;

4. Notons que certains clients FTP permettent le téléchargement d'une arborescence mais cela est opéré en téléchargeant récursivement son contenu, un fichier après l'autre, en utilisant plusieurs commandes de manière transparente pour l'utilisateur.

- **t** : (*table of contents*) l’affichage des entrées contenues dans l’archive (chemins des fichiers et répertoires qu’elle contient) ;
- **r** : l’ajout de fichiers/répertoires dans une archive ;
- **u** : (*update*) la mise à jour des fichiers/répertoires de l’archive. Cela a en fait pour effet d’ajouter à la fin de l’archive les fichiers plus récents que ceux qui y sont déjà contenus. Lors de l’extraction de l’archive, ce sont ces fichiers/répertoires qui seront extraits à la place des plus anciens.

À la suite de l’*action*, on peut spécifier une ou plusieurs *options*. Il existe de nombreuses options. Seules celles les plus couramment utilisées sont décrites ici :

- **f** *référence\_archive* : (*file*) spécifie *référence\_archive* comme l’archive sur laquelle opérer l’*action*. **Cette option est aujourd’hui quasiment toujours employée.** Si elle n’est pas présente, l’*action* sera opérée sur l’archive indiquée par la variable d’environnement **TARPE**. Si cette variable n’existe pas l’*action* sera opérée sur l’entrée ou la sortie standard, selon la nature de l’*action*<sup>5</sup>. Généralement, *référence\_archive* est la référence d’un fichier ordinaire (appelé « **tarfile** », ou « **fichier tar** ») mais ce peut être aussi – pour indiquer l’entrée standard (actions **x** et **t**) ou la sortie standard (actions **c**, **r** et **u**), ou un périphérique.

**i** En réalité, l’option **f** a la syntaxe suivante : **f** [ [ [ *utilisateur* ] @ ] *hôte* : ] *référence\_archive* car on peut demander à **tar** d’opérer une opération sur un *hôte* (ordinateur) distant. C’est une possibilité rarement employée, qui peut poser des problèmes de sécurité.

- **z** : demande une (dé)compression par **gzip** de ce qui est archivé ou extrait ;
- **Z** : demande une (dé)compression par **compress** ;
- **j** : demande une (dé)compression par **bzip2**. Cette option n’existe que pour les versions récentes de **tar** (du projet GNU) ;
- **C** *chemin\_répertoire* : demande de changer de répertoire pour opérer l’*action* ;
- **p** : lors d’une extraction, préserver les permissions des fichiers de l’archive. Sans cette option, les fichiers extraits sont soumis au masque de **umask** ;
- **v** : (*verbose*) mode verbeux, où **tar** affiche les références des fichiers traités pour les actions **c**, **x**, **r** et **u**, ou plus de renseignements sur les fichiers contenus dans l’archive, pour action **t**. Notons qu’en utilisant **2 v**, on obtient encore plus de renseignements pour les actions **c**, **x**, **r** et **u**.

Après les options, on peut spécifier un ou plusieurs arguments *référence*, désignant les fichiers ou répertoires (arborescences) pour lesquels on veut réaliser l’action (archivage, extraction, ajout, mise à jour ou consultation).



**Les options **f** et **C** prenant un argument, elles doivent figurer à la fin des autres options. Si elles sont toutes les deux utilisées, il faut les indiquer séparément.**

**Par exemple,**

```
tar -cvf archive.tar rep
```

**est une bonne utilisation pour archiver l’arborescence de rep dans le fichier archive.tar alors que**

```
tar -cfv archive.tar rep
```

**est incorrect (signifierait archiver dans le fichier v, le fichier archive.tar et l’arborescence de rep).**

**Aussi, pour combiner **f** et **C**, on peut écrire :**

```
tar -cvf archive.tar -C rep rep-à-archiver
```

5. Selon la version de **tar** (notamment sur les anciens systèmes), le comportement par défaut en l’absence de l’option **f** peut être d’utiliser le premier lecteur de bande magnétique, ce qui correspond à un périphérique du genre `/dev/rmt0`.

## Exemples

```
$ tar -czf archive.tgz unix
```

⇒ archivage avec compression du repertoire *unix*, en créant le fichier *archive.tgz*

```
$ tar -xzf archive.tgz -C ../ailleurs
```

⇒ extraction du contenu de l'archive compressée *archive.tgz* dans le repertoire *../ailleurs*. Sans utiliser l'option **-C**, il aurait fallu écrire la commande suivante pour arriver au même résultat (où le repertoire de travail est *ici*) :

```
cd ../ailleurs; tar -xvf ../ici/archive.tgz; cd -
```

```
$ tar -cvf archive.tar unix tpres
```

```
unix/
unix/cigale.txt
unix/amphigouri.txt
unix/cig.txt
unix/ici/
unix/ici/ts_machin.txt
unix/ici/ts_oui.adb
unix/ici/ts_aussi.ads
... ..
tpres/
tpres/tp1/
tpres/tp1/corrigesIV.1.A/
tpres/tp1/corrigesIV.1.A/sendcars.cxx
tpres/tp1/corrigesIV.1.A/Makefile
tpres/tp1/corrigesIV.1.A/INCLUDE_H
... ..
```

⇒ archivage en mode verbeux des repertoires *unix* et *tpres*, en créant l'archive *archive.tar*

```
$ tar -xvf archive.tar -C autrepert unix/bart.gif
```

```
unix/bart.gif
```

⇒ extraction dans le repertoire *autrepert* du fichier *unix/bart.gif* de l'archive *archive.tar* (ce qui crée le repertoire *unix* dans *autrepert*)

```
$ tar -cf - -C repsource . | tar -xpf - -C repcible
```

⇒ crée une copie de *represource* dans le repertoire *repcible* (qui doit exister). En effet, à gauche du pipe, l'archive est créée sur la sortie standard (option **-f -**) qui est extraite à droite du pipe par l'entrée standard (option **-f -**).

□

## 13.D.2 zip/unzip : un archiviste récent et compatible WinZIP

Cet archiviste est en réalité compatible avec PKZIP, un archiviste que l'on trouve depuis longtemps sur MS-DOS et sur Windows. WinZIP utilise encore aujourd'hui cet algorithme mais il semblerait que ce ne sera bientôt plus le cas.

Quoi qu'il en soit, pour le moment, les archives créées avec WinZIP et portant l'extension *.zip* sont exploitables par les utilitaires **zip** et **unzip** de Linux. Consulter le manuel en ligne pour en connaître les nombreuses utilisations possibles.

# Chapitre 14

## Compléments sur les fichiers

---

### 14.A ln : créer un lien

Sur Unix, il existe deux types de liens : les **liens physiques** (appelés aussi liens matériels ou liens durs) et les **liens symboliques**. Ces liens sont fondamentalement différents mais sont tous deux créés par la commande externe **ln** (*link*).

#### Synopsis

```
ln [-sfv] source [destination]
```

```
ln [-sfv] source {source} répertoire
```

Sans l'option **-s**, chaque *source* doit être une référence à un fichier ordinaire. Si *destination* n'est pas spécifiée, **ln** crée un lien portant le même nom que *source* dans le répertoire de travail. Sinon, **ln** crée un lien *destination* sur *source*.

Si plusieurs *sources* sont spécifiées, *répertoire* doit exister et un lien pour chaque *source* y sera créé, avec le même nom que celui de la *source* correspondante.

Par défaut, **ln** ne remplace pas les fichiers existants, à moins que l'option **-f** soit spécifiée. Les liens créés sont par défaut des liens physiques. L'option **-s** demande la création de lien(s) symbolique(s).

L'option **-v** active le mode verbeux (un message est affiché pour chaque lien créé).

#### 14.A.1 Lien physique

Un lien physique est simplement un autre nom pour le même fichier que la *source*. Pour le comprendre, il faut préciser ce qu'est un nom de fichier et introduire la notion d'**inode**.


#### Les inodes


En fait, un répertoire n'est autre qu'un fichier particulier qui contient une liste d'associations <*nom\_de\_fichier*, *inode*>. Lorsqu'on désigne un nom de fichier, le système recherche l'inode auquel il est associé dans le répertoire correspondant.

Le nombre d'inodes d'un système de fichiers (une partition) est limité. Il détermine le nombre de fichiers (de tout type) que peut contenir la partition. La table des inodes est un point d'entrée vers tous les fichiers d'une


partition. Chaque inode possède un numéro qui est son index dans la table. Un inode est une structure<sup>1</sup> qui est le véritable point d'entrée d'un fichier dans un système de fichiers. Lorsqu'un fichier (de tout type) est créé, un inode lui est attribué et contient les informations suivantes :

- son type : fichier ordinaire, répertoire, etc.
- l'UID de son propriétaire
- le GID de son groupe
- ses permissions
- son nombre de liens physiques
- sa taille en octets
- la date de dernière modification du fichier (appelée *mtime*)
- la date de dernière modification de l'inode (appelée *ctime*)
- la date de dernier accès en lecture au fichier (appelée *atime*)
- diverses informations parmi lesquelles des pointeurs sur le contenu du fichier

 Un inode contient donc tout ce qui concerne un fichier, **sauf son nom !**

 La commande externe **stat** permet d'obtenir (une partie) des informations contenues dans l'inode d'un fichier (au sens large). Elle permet aussi d'obtenir des informations sur le système de fichiers auquel un fichier appartient.

Un nom de fichier n'est donc qu'une façon de désigner un inode (et donc un fichier). Il n'est pas étonnant qu'un inode puisse être désigné par plusieurs noms, ces noms pouvant d'ailleurs être contenus dans des répertoires différents. La seule restriction est que les noms doivent être contenus dans le même système de fichiers (même partition d'un même disque). L'inode contient un compteur du nombre de noms qui le désignent, c'est à dire son nombre de liens physiques. Un fichier n'est réellement supprimé (et son inode rendu disponible) que lorsque plus aucun nom ne le désigne (et qu'aucun processus n'utilise le fichier).

 Lorsqu'on crée un lien physique à partir d'un (nom de) fichier, le nom d'origine du fichier a exactement le même statut que le nouveau nom : le système ne garde pas de trace de quel nom était celui d'origine.

## Les liens physiques et les commandes **rm**, **mv** et **ls**

Lorsqu'on utilise la commande **rm**, on ne fait que supprimer un lien physique pour le fichier, c'est à dire une association *<nom\_de\_fichier, inode>*, et décrémenter le nombre de liens physiques pour l'inode. Le fichier (et l'inode) n'est supprimé que si ce nombre passe à 0 (et qu'aucun processus ne l'utilise à cet instant).

La commande **mv**, quant à elle, n'effectue pas la même opération selon que la destination se trouve sur le même système de fichier que la source ou pas :

- si oui, elle crée un lien physique pour le nouveau nom, ce qui ne coûte rien (ou pas grand chose)
- sinon, elle copie entièrement la source dans le système de fichier destination. C'est un cas particulier du déplacement, car il faut avoir le droit de lecture sur le fichier d'origine

1. Une structure est un ensemble d'informations de différents types.

dans les deux cas, elle supprime ensuite le lien physique de la source.

L'option **-i** de **ls** demande l'affichage du numéro d'inode. Le nombre de liens physiques d'un fichier est indiqué avec les informations détaillées demandées avec l'option **-l** de **ls**.

### Exemples

```
$ ls -l
total 4
-rw-r--r-- 1 cyril users 58 2008-02-08 16:52 unfic
```

⇒ le fichier *unfic* n'a qu'un lien physique, indiqué par le chiffre qui suit les permissions

```
$ stat unfic
  File: 'unfic'
  Size: 58          Blocks: 8          IO Block: 4096   fichier régulier
Device: 306h/774d   Inode: 33947        Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/   cyril)   Gid: (  100/   users)
Access: 2008-02-08 16:41:55.000000000 +0100
Modify: 2008-02-08 16:52:25.000000000 +0100
Change: 2008-02-08 16:52:25.000000000 +0100
```

⇒ **stat** affiche des informations sur le fichier et son inode

```
$ ln unfic autre_nom
```

⇒ création d'un lien physique s'appelant *autre\_nom* pour le même fichier que *unfic*

```
$ ls -l
total 8
-rw-r--r-- 2 cyril users 58 2008-02-08 16:52 autre_nom
-rw-r--r-- 2 cyril users 58 2008-02-08 16:52 unfic
```

⇒ les deux fichiers ont maintenant deux liens physiques

```
$ stat unfic
  File: 'unfic'
  Size: 58          Blocks: 8          IO Block: 4096   fichier régulier
Device: 306h/774d   Inode: 33947        Links: 2
Access: (0644/-rw-r--r--)  Uid: ( 1000/   cyril)   Gid: (  100/   users)
Access: 2008-02-08 16:41:55.000000000 +0100
Modify: 2008-02-08 16:52:25.000000000 +0100
Change: 2008-02-08 16:54:02.000000000 +0100
```

⇒ on voit que la date *Change* (ctime) a changé du fait de la mise à jour du nombre de liens

```
$ ls -li
total 8
33947 -rw-r--r-- 2 cyril users 58 2008-02-08 16:52 autre_nom
33947 -rw-r--r-- 2 cyril users 58 2008-02-08 16:52 unfic
```

⇒ on peut combiner les options **-i** et **-l** de **ls** et le numéro d'inode est affiché en début de ligne

```
$ chmod 755 autre_nom
```

⇒ *modification des permissions de autre\_nom*

```
$ ls -l
```

```
total 8
```

```
-rwxr-xr-x 2 cyril users 58 2008-02-08 16:52 autre_nom
```

```
-rwxr-xr-x 2 cyril users 58 2008-02-08 16:52 unfic
```

⇒ *les permissions de unfic ont aussi changé : les permissions sont relatives à l'inode et pas à un nom de fichier*

```
$ cat unfic
```

je suis le contenu d'un fichier on ne peut plus ordinaire

```
$ echo 'salut tout le monde...' > autre_nom
```

```
$ cat unfic
```

```
salut tout le monde...
```

⇒ *en modifiant autre\_nom, on modifie aussi unfic*

```
$ rm unfic
```

```
$ ls -l
```

```
total 4
```

```
-rwxr-xr-x 1 cyril users 23 2008-02-08 16:54 autre_nom
```

⇒ *en supprimant unfic, autre\_nom n'a plus qu'un lien physique*

```
$ cat autre_nom
```

```
salut tout le monde...
```

⇒ *la suppression de unfic n'a aucune incidence sur autre\_nom qui continue d'exister*

□

## Liens physiques et répertoires

Le système n'autorise pas la création de liens physiques sur des répertoires, même pour root<sup>2</sup>. Cependant, le système en gère de lui-même. En effet, lorsqu'on crée un répertoire, il possède deux liens : son nom dans le répertoire qui le contient, et le nom . (point) qu'il contient lui-même. Ainsi, le nombre de liens indiqué par l'option **-l** de **ls** a aussi un sens pour les répertoires.

### Exemple

```
$ mkdir rep
```

```
$ ls -ld rep
```

```
drwxr-xr-x 2 bart simpson 4096 jan 12 09:10 rep
```

⇒ *le nombre de liens d'un répertoire nouvellement créé est toujours 2*

```
$ mkdir rep/autre_rep
```

⇒ *création de autre\_rep dans rep. Du coup, rep devrait être aussi désigné par le répertoire . . contenu dans autre\_rep ?*

```
$ ls -ld rep
```

```
drwxr-xr-x 3 bart simpson 4096 jan 12 09:10 rep
```

⇒ *Oui !*

□

2. L'option **-d** de **ln** est prévue pour ça, mais elle n'est pas fonctionnelle.



## D'accord, mais à quoi sert un lien physique ?

L'utilisation la plus courante d'un lien physique concerne les fichiers exécutables. Cela permet au même code exécutable d'avoir plusieurs noms sans occuper d'espace disque supplémentaire. Il peut alors adapter son comportement selon le nom par lequel il a été appelé.

Les commandes **gzip**, **gunzip** et **zcat** (vues plus loin) sont sur une Mandriva<sup>3</sup> 2007 des noms désignant le même inode. Le code correspondant adapte son comportement selon le nom par lequel il a été appelé, et n'admet pas les mêmes options.

### Exemple (sur une Mandriva 2007)

```
$ ls -l /bin/gzip /bin/gunzip /bin/zcat
-rwxr-xr-x  3 root root 49584 mai 25  2005 /bin/gunzip*
-rwxr-xr-x  3 root root 49584 mai 25  2005 /bin/gzip*
-rwxr-xr-x  3 root root 49584 mai 25  2005 /bin/zcat*
```

⇒ ces fichiers ont 3 liens physiques


```
$ ls -li /bin/gzip /bin/gunzip /bin/zcat
2064 -rwxr-xr-x  3 root root 49584 mai 25  2005 /bin/gunzip*
2064 -rwxr-xr-x  3 root root 49584 mai 25  2005 /bin/gzip*
2064 -rwxr-xr-x  3 root root 49584 mai 25  2005 /bin/zcat*
```

⇒ les trois noms désignent le même inode !!

□

## 14.A.2 Lien symbolique


À la différence d'un lien physique qui est un fichier de même type (le plus souvent ordinaire) que le fichier *source* à partir duquel il a été créé, **un lien symbolique est un fichier spécial** qui contient le chemin du fichier qu'il "pointe".

 Un lien symbolique joue le même rôle que les raccourcis de Windows (fichiers d'extension `.lnk`).

C'est donc un fichier qui désigne un autre nom de fichier, qui peut être de n'importe quel type.

En utilisant l'option **-s** de **ln**, on crée un lien symbolique contenant *source*. La taille d'un lien symbolique est le nombre d'octets de la chaîne *source*. **ln** ne vérifie pas que *source* existe. De plus, pour que le lien soit correct, il faut que *source* soit une référence valide du fichier cible depuis le répertoire contenant le lien (et non pas depuis le répertoire de travail lorsque l'on tape la commande).

 Avec l'option **-l** de **ls**, un lien symbolique apparaît comme un fichier spécial de type **l**.

 Une différence très importante avec les liens physiques et qu'un lien symbolique "pointe" sur un chemin (relatif ou absolu) d'un fichier (au sens large). Si le fichier "pointé" est supprimé, déplacé ou renommé, alors le lien symbolique devient orphelin (il ne désigne plus aucun fichier).

3. ce n'est pas le cas sur une Debian où, à ce jour, **gunzip** et **uncompress** désignent le même inode.

En contrepartie, un lien symbolique peut être fait d'un système de fichiers à un autre, ou sur un répertoire. D'autre part, un lien symbolique n'a pas de permissions propres : ce sont celles (du chemin) de la cible.

Certaines commandes admettent une option permettant de déréférencer un lien symbolique, de manière à ce que la commande s'applique au fichier pointé et non au lien. Pour **ls**, cette option est **-H**.

### Exemple

```
$ ls -l
total 4
drwxr-xr-x 2 cyril users 4096 2008-02-11 08:13 rep
-rw-r--r-- 1 cyril users 23 2008-02-08 16:54 unfic
```

```
$ ln -s unfic symb1
```

⇨ création du lien symbolique *symb1* vers *unfic*

```
$ ln -s ../fic1 symb2
```

⇨ création du lien symbolique *symb2* vers *../fic1*

```
$ ln -s /etc/passwd symb3
```

⇨ création du lien symbolique *symb3* vers */etc/passwd*

```
$ ln -s rep symb4
```

⇨ création du lien symbolique *symb4* vers le répertoire *rep*

```
$ ls -l
total 4
drwxr-xr-x 2 cyril users 4096 2008-02-11 08:13 rep
lrwxrwxrwx 1 cyril users 5 2008-02-11 08:05 symb1 -> unfic
lrwxrwxrwx 1 cyril users 7 2008-02-11 08:06 symb2 -> ../fic1
lrwxrwxrwx 1 cyril users 11 2008-02-11 08:06 symb3 -> /etc/passwd
lrwxrwxrwx 1 cyril users 3 2008-02-11 08:13 symb4 -> rep
-rw-r--r-- 1 cyril users 23 2008-02-08 16:54 unfic
```

⇨ on voit que *symb1* à *symb4* sont des fichiers spéciaux de type **1**. On remarque aussi que le nombre de liens physiques de *unfic* est resté à 1. Aussi, la taille de *symb1* à *symb4* est exactement le nombre d'octets du chemin pointé.

```
$ echo 'hello' > symb1
```

⇨ on écrase le contenu du fichier pointé par *symb1* (*unfic*)

```
$ cat unfic
```

```
hello
```

```
$ echo 'hello' > symb3
```

```
-bash: symb3: Permission non accordée
```

⇨ tentative avortée d'écrasement du fichier pointé par *symb3* (*/etc/passwd*)

```
$ rm unfic
```

```
$ ls -l symb1
```

```
lrwxrwxrwx 1 cyril users 5 2008-02-11 08:05 symb1 -> unfic
```

⇨ après la suppression de *unfic*, *symb1* existe toujours mais est orphelin (il devrait être affiché avec une couleur différente, souvent en rouge sur fond noir)

```
$ cat symb1
```

```
cat: symb1: Aucun fichier ou répertoire de ce type
```

⇒ *symb1 étant orphelin, on ne peut afficher son contenu*

```
$ ls -lH symb1
```

```
ls: symb1: Aucun fichier ou répertoire de ce type
```

⇒ *en déréférençant symb1, ls indique que le fichier pointé (unfic) n'existe pas*

```
$ ls -lH symb3
```

```
-rw-r--r-- 1 root root 1513 2007-09-25 07:29 symb3
```

⇒ *en déréférençant symb3, ls indique les détails sur le fichier pointé par symb3 (/etc/passwd)*

```
$ ln -s rep symb1
```

```
ln: création d'un lien symbolique 'symb1' vers 'rep': Le fichier existe.
```

⇒ *ln refuse d'écraser le fichier symb1*

```
$ ln -sf rep symb1
```

⇒ *l'option -f le force à le faire*

```
$ ls -l symb1
```

```
lrwxrwxrwx 1 cyril users 3 2008-02-11 08:31 symb1 -> rep
```

⇒ *maintenant, symb1 est un lien symbolique vers le répertoire rep*

```
$ mv symb2 symb1
```

⇒ *déplacement de symb2 dans le répertoire pointé par symb1. Or symb2 était un lien symbolique relatif vers ../fic1. Il devient alors orphelin.*

```
$ mv symb3 symb1
```

⇒ *déplacement de symb3 dans le répertoire pointé par symb1. Puisque symb3 était un lien symbolique absolu vers /etc/passwd, cette opération ne le rend pas orphelin.*

□

## 14.B Compléments sur les permissions

### 14.B.1 Changer d'identité en exécutant un programme

#### 14.B.1.a Le bit set-uid sur les fichiers

Certains outils nécessitent d'être exécutés avec des privilèges particuliers. Par exemple, un utilisateur normal n'a pas le droit de modifier le fichier `/etc/passwd` ni le fichier `/etc/group` : seul root peut le faire. Cependant, en utilisant les commandes **passwd**, **chfn** et **chsh**, n'importe quel utilisateur peut changer son mot de passe, son intitulé et le shell qu'il souhaite utiliser, ce qui modifie directement le fichier `/etc/passwd`<sup>4</sup>. De même, un utilisateur quelconque qui serait administrateur d'un groupe peut en modifier les membres en utilisant la commande **gpasswd**, ce qui modifie le fichier `/etc/group`.

Cela n'est possible que parce que ces commandes ont le **bit set-uid** positionné. De ce fait, lorsqu'un utilisateur les exécute, le processus correspondant prend l'identité du propriétaire du fichier, soit root !!

En fait, tout processus possède deux identités d'utilisateur : l'utilisateur réel et l'utilisateur effectif. Les permissions accordées à un processus sont déterminées en fonction de son utilisateur effectif.

4. Pour des raisons de sécurité, les mots de passe sont généralement stockés dans le fichier `/etc/shadow` qui n'est même pas lisible par un autre utilisateur que root mais qui est modifié lorsque l'utilisateur change son mot de passe.

✍ On reconnaît un fichier qui a le *bit set-uid* positionné avec les informations détaillées de **ls**, où à la place du **x** pour le propriétaire, il y a **s** ou **S**. S'il y a **S**, c'est que le propriétaire n'a pas le droit d'exécution sur le fichier.

✍ Pour des raisons de sécurité, le bit set-uid est ignoré pour certains exécutables, tels que les scripts.

## Exemples

On dispose ici de quatre copies du même programme *u-infos1*, *u-infos2*, *u-infos3* et *u-infos4* indiquant l'utilisateur réel et l'utilisateur effectif du processus qui l'exécute. Les trois exécutables sont propriétés de *cpb*. Ils diffèrent simplement par leurs permissions.

### Utilisateur *cpb*

```
$ ls -l u-infos*
```

```
-r-xr-xr-x    1 cpb      prof      14293  jan   2 17:09 u-infos1
-r-Sr-xr-x    1 cpb      prof      14293  jan   2 17:09 u-infos2
-r-sr-xr-x    1 cpb      prof      14293  jan   2 17:09 u-infos3
-r-sr--r--    1 cpb      prof      14293  jan   2 17:16 u-infos4
```

⇒ *u-infos2*, *u-infos3* et *u-infos4* ont le bit set-uid positionné. *u-infos2* n'est pas exécutable par son propriétaire, et *u-infos4* n'est pas exécutable par les membres du groupe et les autres.

```
$ ./u-infos1
```

```
Ce processus a pour utilisateur réel      : 5778 (cpb)
                  et pour utilisateur effectif : 5778 (cpb)
```

```
$ ./u-infos2
```

```
./u-infos2: Permission denied.
```

```
$ ./u-infos3
```

```
Ce processus a pour utilisateur réel      : 5778 (cpb)
                  et pour utilisateur effectif : 5778 (cpb)
```

```
$ ./u-infos4
```

```
Ce processus a pour utilisateur réel      : 5778 (cpb)
                  et pour utilisateur effectif : 5778 (cpb)
```

⇒ Le propriétaire ne peut pas exécuter *u-infos2* mais pour le reste, ça ne change rien. En effet, le bit set-uid ne sert que pour les utilisateurs non propriétaires du fichier.

### Utilisateur *bart*

```
$ whoami
```

```
bart
```

⇒ la commande **whoami** permet de connaître l'identité (effective) de l'utilisateur en session.

```
$ groups
```

```
simpson
```

```
$ ./u-infos1
```

```
Ce processus a pour utilisateur réel      : 6015 (bart)
                  et pour utilisateur effectif : 6015 (bart)
```

⇒ l'utilisateur *bart* exécute *u-infos1* qui n'a pas le bit set-uid positionné : le processus possède donc son identité

```
$ ./u-infos2
```

```
Ce processus a pour utilisateur réel      : 6015 (bart)
                  et pour utilisateur effectif : 5778 (cpb)
```

⇒ *en revanche, u-infos2 a le bit set-uid positionné, donc l'utilisateur effectif du processus est cpb (alors que cpb n'a pas le droit de l'exécuter !)*

```
$ ./u-infos3
```

```
Ce processus a pour utilisateur réel      : 6015 (bart)
                  et pour utilisateur effectif : 5778 (cpb)
```

⇒ *De même pour u-infos3*

```
$ ./u-infos4
```

```
./u-infos4: Permission denied.
```

⇒ *En revanche, le bit set-uid est positionné pour u-infos4 mais bart n'a pas les droits d'exécution.*

□

 Tout fichier créé par un processus est propriété de l'utilisateur effectif de ce processus !

### Exemple

```
$ ls -l ficreation
```

```
---s--x--x    1    cpb    prof            14429 jan  3 15:20 ficreation
```

⇒ *l'exécutable ficreation crée un fichier dont le nom est passé en argument et contenant un texte indiquant quel est l'utilisateur réel qui l'a créé. Il a le bit set-uid positionné.*

```
$ whoami
```

```
bart
```

⇒ *l'utilisateur en session est bart*

```
$ ./ficreation bidule
```

⇒ *L'utilisateur bart exécute ficreation pour créer le fichier bidule*

```
$ ls -l bidule
```

```
-rw-r--r--    1    cpb    simpson          22 jan  3 15:21 bidule
```

⇒ *on voit bien que le propriétaire du fichier créé par bart est cpb !*

```
$ cat bidule
```

```
Ce fichier a été créé par l'utilisateur réel 6015 (bart)
```

□

❗ Il existe des fonctions du langage C (et C++) permettant d'obtenir les numéros d'utilisateur réel et effectif du processus appelant. Leur prototype est le suivant :

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

uid_t getuid(void);
uid_t geteuid(void);
```

### 14.B.1.b Le bit set-gid sur les fichiers

De façon analogue au bit set-uid, il existe le **bit set-gid**. S'il est positionné pour un exécutable alors le processus qui l'exécute a pour groupe effectif le groupe propriétaire du fichier. Le processus a aussi un groupe réel qui est celui de l'utilisateur en session.



**Ce bit est ignoré si les membres du groupe n'ont pas le droit d'exécution sur le fichier.**



On reconnaît un fichier qui a le *bit set-gid* positionné avec les informations détaillées de **ls**, où à la place du **x** pour le groupe, il y a **s** ou **S**. S'il y a **S**, c'est que les membres du groupe n'ont pas le droit d'exécution sur le fichier.

Par exemple, l'utilitaire **write** a le bit set-gid positionné. Il est du groupe tty comme la plupart des terminaux des utilisateurs. Il permet ainsi à n'importe quel utilisateur d'écrire un message sur le terminal d'un autre utilisateur qui a positionné les droits d'écriture pour le groupe tty sur son terminal en utilisant la commande **mesg**.



Pour des raisons de sécurité, le bit set-gid est ignoré pour les scripts shell.

#### Exemple

```
$ whoami
bart
$ groups
simpson
$ ls -l g-infos
---x--s--x  1 cpb      prof          14075 jan  3 14:27 g-infos
```

⇒ Le fichier *g-infos* est exécutable par tous et a le bit set-gid positionné

```
$ ./g-infos
Ce processus a pour groupe réel      : 504 (simpson)
et pour groupe effectif : 20 (prof)
```

⇒ Ainsi, si l'utilisateur bart du groupe simpson l'exécute, le processus créé a prof pour groupe effectif

□



Tout fichier créé par un processus a pour groupe le groupe effectif de ce processus !



Il existe des fonctions du langage C (et C++) permettant d'obtenir les numéros de groupe réel et effectif du processus appelant. Leur prototype est le suivant :

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

gid_t getgid(void);
gid_t getegid(void);
```

### 14.B.1.c Combinaison des bits set-uid et set-gid sur les fichiers

Si un utilisateur exécute un fichier qui a les bits set-uid et set-gid positionnés, alors le processus créé aura pour utilisateur et groupe effectifs ceux du fichier.

#### Exemple

```
$ whoami
bart
$ ls -l ug-infos
---s---s--x  1 cpb      prof      14741 jan  3 14:27 ug-infos
```

⇒ Le fichier *ug-infos* a les deux bits set-uid et set-gid positionnés et est exécutable par tous.

```
$ ./ug-infos
Ce processus a pour utilisateur réel      : 6015 (bart)
                  et pour groupe réel      :  504 (simpson)
                  et pour utilisateur effectif : 5778 (cpb)
                  et pour groupe effectif  :   20 (prof)
```

⇒ Lorsque bart du groupe simpson l'exécute, le processus correspondant a cpb et prof pour utilisateur et groupe effectifs.

□

## 14.B.2 Permissions supplémentaires sur les répertoires

### 14.B.2.a Le bit set-gid sur les répertoires

Le bit set-gid a aussi un sens sur les répertoires. S'il est activé sur un répertoire, alors tout fichier créé dans ce répertoire aura pour groupe celui de ce répertoire, même si l'utilisateur qui a créé ce fichier n'est pas de ce groupe (il faut tout de même qu'il ait le droit de créer un fichier dans ce répertoire). Cela s'avère utile pour travailler sur un projet avec un groupe de personnes.

Si en plus le fichier créé est de type répertoire, alors le bit set-gid est aussi activé sur le nouveau répertoire.

#### Exemples

```
$ whoami
toto
$ groups
toto etud1
$ ls -ld parici
drwxr-srwx  2 bob      prof      4096 aoû 30 16:22 parici
$ cd parici
$ echo 'abc' > fic1
$ mkdir repdetoto
```

⇒ l'utilisateur toto a créé *fic1* et *repedetoto* dans le repertoire *parici* de bob qui a le bit set-gid activé et qui est du groupe prof auquel n'appartient pas toto

```
$ ls -l
total 8
-rw-r--r--  1 toto      prof      4 aoû 30 16:24 fic1
drwxr-sr-x  2 toto      prof      4096 aoû 30 16:24 repdetoto
```

⇒ *fic1* et *repedetoto* sont du groupe prof, et *repedetoto* a lui aussi son bit set-gid positionné

□

### 14.B.2.b Le bit set-uid sur les répertoires

Ce bit peut aussi être activé sur les répertoires. Sur certains systèmes, il devrait jouer le même rôle que le bit set-gid, c'est à dire qu'un fichier créé dans un répertoire ayant ce bit positionné ait comme propriétaire celui du répertoire.



**Cependant, je n'ai jamais rencontré un système qui l'interprète de cette manière. Sur Linux, il est ignoré.**

### 14.B.2.c Le sticky-bit

Il existe encore un bit pour les permissions des fichiers : le « **sticky-bit** ». Sa signification est quelque peu différente d'une version d'Unix à une autre. Pour Linux, il ne sert que pour les répertoires et indique que seuls le propriétaire d'un répertoire peut supprimer un fichier qui s'y trouve, ou le propriétaire de ce fichier (s'il a les droits d'écriture et d'exécution sur ce répertoire). Évidemment, root a toujours la possibilité de supprimer n'importe quel fichier ou répertoire.

En effet, sans *sticky-bit*, si un utilisateur a la permission d'écriture et d'exécution sur un répertoire, alors il peut ajouter des fichiers dans ce répertoire mais aussi supprimer n'importe quel fichier qu'il contient, même si ce fichier ne lui appartient pas et même s'il n'a pas le droit d'écriture sur ce fichier !!!



On reconnaît un répertoire (ou fichier) qui a le sticky-bit positionné avec les informations détaillées de **ls**, où à la place du **x** pour les autres, il y a **t** ou **T**. S'il y a **T**, c'est que les autres n'ont pas le droit d'exécution sur le fichier.



Sur tous les Linux, le répertoire `/tmp` est accessible et lecture/écriture/exécution pour tous les utilisateurs. Il contient des fichiers temporaires créés par divers processus et appartenant à divers utilisateurs, le plus souvent sans que ces derniers ne le sachent. Le sticky-bit est activé sur ce répertoire pour éviter qu'un utilisateur ne supprime un fichier d'un autre utilisateur.

## Exemples

### Utilisateur homer

```
$ whoami
homer
$ ls -ld
drwxrwxrwx    2 cpb      prof      4096 jan  3 17:55 .
```



*le répertoire de travail appartient à l'utilisateur cpb du groupe prof. Tout le monde a le droit d'écriture dans ce répertoire*

```
$ ls -l
total 12
-rw-r--r--    1 cpb      prof      11 jan  3 17:55 fichier-1
-rw-r--r--    1 bart     simpson   17 jan  3 17:29 fichier-2
-rw-r--r--    1 homer    simpson   18 jan  3 17:51 fichier-3
```



*Il contient 3 fichiers, chacun appartenant à un utilisateur différent*



```
$ rm fichier-1
```

```
rm: détruire un fichier protégé en écriture fichier régulier `fichier-1'? y
```

```
$ ls -l
```

```
total 8
```

```
-rw-r--r--  1 bart      simpson      17 jan  3 17:29 fichier-2
-rw-r--r--  1 homer     simpson      18 jan  3 17:51 fichier-3
```

⇒ *L'utilisateur homer a pu supprimer fichier-1 alors que ce fichier ne lui appartenait pas et qu'il n'avait même pas le droit d'écriture dessus. C'est normal car homer a le droit d'écriture sur le répertoire de travail (qui ne lui appartient pas non plus).*

### Utilisateur cpb

```
$ chmod 1777 .
```

⇒ *Le propriétaire du répertoire de travail active le sticky-bit*

```
$ ls -ld
```

```
drwxrwxrwt  2 cpb      prof          4096 jan  3 17:59 .
```

⇒ *on voit que le sticky-bit est effectivement activé par le **t** des permissions*

### Utilisateur homer

```
$ rm fichier-2
```

```
rm: détruire un fichier protégé en écriture fichier régulier `fichier-2'? y
```

```
rm: Ne peut enlever `fichier-2': Operation not permitted
```

⇒ *Cette fois homer ne peut pas supprimer fichier-2 car il ne lui appartient pas et homer n'est pas non plus propriétaire du répertoire de travail*

```
$ rm fichier-3
```

⇒ *En revanche, homer a le droit de supprimer ses propres fichiers*

### Utilisateur cpb

```
$ rm fichier-2
```

```
rm: détruire un fichier protégé en écriture fichier régulier `fichier-2'? y
```

```
$ ls -l
```

```
total 0
```

⇒ *Le propriétaire du répertoire de travail peut supprimer fichier-2 même s'il ne lui appartient pas*

□

## 14.B.3 chmod pour modifier les bits set-uid, set-gid et sticky

Pour positionner ou enlever les bits set-uid, set-gid ou sticky, il faut utiliser la commande **chmod**.

### 14.B.3.a Mode absolu

Au chapitre 2, nous avons dit que le mode absolu de **chmod** s'exprimait avec 3 chiffres  $C_u C_g C_o$ . En réalité, il s'exprime avec 4 chiffres  $C_b C_u C_g C_o$ . En effet, le premier chiffre peut être omis s'il vaut 0 (zéro). Concrètement, les commandes suivantes sont équivalentes :

```
chmod 754 fichier
```

```
=
```

```
chmod 0754 fichier
```

❗ En fait, les premiers chiffres peuvent être omis sauf  $C_o$ . Ils sont alors considérés à 0. La syntaxe exacte est plutôt

$$[[[C_b]C_u]C_g]C_o$$

C'est le premier chiffre  $C_b$  qui permet de modifier les bits set-uid, set-gid et sticky. Il se calcule comme une somme où l'on ajoute :

- 4 pour le bit set-uid ;
- 2 pour le bit set-gid ;
- 1 pour le sticky-bit.

### Exemples

```
$ ls -l
total 8
drwxrwxrwx    2 cpb      prof      4096 jan  4 11:05 rep
-rw-r-xr-x    1 cpb      prof      11 jan  4 11:05 un-fic
$ chmod 4655 un-fic
```

⇒ En plus des permissions existantes, positionne le bit set-uid pour *un-fic*

```
$ ls -l un-fic
-rwSr-xr-x    1 cpb      prof      11 jan  4 11:05 un-fic
```

⇒ On voit le **S** (cpb n'a pas le droit d'exécution sur *un-fic*)

```
$ chmod 2655 un-fic
```

⇒ Enlève le bit set-uid mais ajoute le bit set-gid sur *un-fic*

```
$ ls -l un-fic
-rw-r-sr-x    1 cpb      prof      11 jan  4 11:05 un-fic
```

⇒ On voit le **s** correspondant (les membres du groupe ayant le droit d'exécution sur *un-fic*)

```
$ chmod 6655 un-fic
```

⇒ Positionne à la fois le bit set-uid et le bit set-gid sur *un-fic*

```
$ ls -l un-fic
-rwSr-sr-x    1 cpb      prof      11 jan  4 11:05 un-fic
```

```
$ chmod 1777 rep
```

⇒ Positionne le sticky-bit sur *rep*

```
$ ls -ld rep
drwxrwxrwt    2 cpb      prof      4096 jan  4 11:05 rep
```

⇒ On le voit par le **t** (les autres ont le droit d'exécution sur *rep*)

```
$ chmod 1776 rep
```

⇒ Enlève le droit d'exécution pour les autres sur *rep*

```
$ ls -ld rep
drwxrwxrwt    2 cpb      prof      4096 jan  4 11:05 rep
```

⇒ On voit alors le **T** du sticky-bit mais sans droit d'exécution pour les autres.

□

### 14.B.3.b Mode symbolique

On peut aussi utiliser le mode symbolique pour fixer ces bits. On rappelle que le mode symbolique est décrit par :

$$(qui) (opérateur) (quoi) \{ , (qui) (opérateur) (quoi) \}$$

où *qui* indique à qui on veut fixer des permissions. C'est une combinaison des lettres **u**, **g**, **o** et **a**, représentant :

- u** le propriétaire
- g** le groupe
- o** les autres
- a** tous

L'*opérateur* est l'un des signes **+**, **=** et **-**, pour indiquer si l'on veut :

- +** ajouter
- =** fixer exactement
- supprimer

des permissions.

Enfin, *quoi* indique les permissions concernées par la modification. C'est une combinaison des lettres **r**, **w**, **x**, **s** et **t**, représentant :

- r** la lecture
- w** l'écriture
- x** l'exécution
- s** les bits set-uid et set-gid
- t** le sticky-bit

Ainsi, le caractère **s** représente à la fois le bit set-uid et le bit set-gid : cela dépend de la catégorie d'utilisateur à qui s'applique la permission : **u+s** ajoute le bit set-uid, et **g+s** ajoute le bit set-gid, alors que **ug+s** ou **+s** ajoute à la fois le bit set-uid et le bit set-gid. Le sticky-bit ne peut être ajouté que sur les autres.

### Exemples

On refait les mêmes modifications que dans l'exemple précédent mais avec le mode symbolique, ce qui s'avère souvent plus pratique.

```
$ ls -l
total 8
drwxrwxrwx    2 cpb      prof      4096 jan  4 11:05 rep
-rw-r-xr-x    1 cpb      prof      11 jan  4 11:05 un-fic
$ chmod u+s un-fic
```

⇒ En plus des permissions existantes, ajoute le bit set-uid pour *un-fic*

```
$ ls -l un-fic
-rwSr-xr-x    1 cpb      prof      11 jan  4 11:05 un-fic
$ chmod u-s,g+s un-fic
```

⇒ Enlève le bit set-uid mais ajoute le bit set-gid sur *un-fic*

```
$ ls -l un-fic
-rw-r-sr-x    1 cpb      prof      11 jan  4 11:05 un-fic
$ chmod ug+s un-fic
```

⇨ Ajoute à la fois le bit set-uid et le bit set-gid sur *un-fic* (à ce stade, **u+s** aurait suffi, mais aussi **+s**).

```
$ ls -l un-fic
-rwSr-sr-x    1 cpb      prof      11 jan  4 11:05 un-fic
$ chmod o+t rep
```

⇨ Ajoute le sticky-bit sur *rep* (un simple **+t** a le même effet)

```
$ ls -ld rep
drwxrwxrwt    2 cpb      prof      4096 jan  4 11:05 rep
$ chmod o-x rep
```

⇨ Enlève le droit d'exécution pour les autres sur *rep*

```
$ ls -ld rep
drwxrwxrwt    2 cpb      prof      4096 jan  4 11:05 rep
```

□

# Chapitre 15

## Bash : variables et tableaux

---

Comme tous les shells, bash crée lui-même un certain nombre de variables et tableaux et permet aux utilisateurs d'en créer. Les variables ont plusieurs objectifs :

- écrire des programmes shell (appelés *shell-scripts*) car bash est aussi un langage de programmation ;
- maintenir des renseignements sur l'environnement du processus en cours, que l'utilisateur ou les utilitaires peuvent exploiter ;
- indiquer des souhaits pour le comportement par défaut de certains utilitaires.

Dans les deux derniers cas, on utilise des variables qui ont un statut particulier : les variables d'environnement.

**i** Nous avons déjà succinctement présenté des variables : les paramètres positionnels (voir section 9.B) et la variable d'environnement **PATH**.

Une variable ou un tableau a un **identificateur** qui est une suite de caractères alphanumériques et d'*underscores* (`_`). Selon la version, les caractères accentués sont autorisés. Un identificateur ne doit pas commencer par un chiffre, sauf pour les paramètres positionnels qui sont créés par bash. Enfin, l'identificateur de variable `_` est réservé pour bash.

### 15.A Les variables de type chaîne

Comme pour la plupart des shells existants, **une variable que l'on crée est par défaut de type chaîne** : c'est à dire que son contenu est stocké comme une chaîne de caractères. Cela ne l'empêche pas de contenir une valeur numérique comme 10, mais son contenu sera stocké comme la chaîne "10". Cependant, selon le contexte de son utilisation, notamment dans une expression arithmétique, son contenu sera traité comme le nombre 10.

bash permet aussi la création de variables entières, qu'il faut déclarer explicitement comme telles, et qui ne peuvent contenir que des entiers. Cette possibilité ne sera pas étudiée dans ce cours car on peut se passer aisément de ce type de variables.


#### 15.A.1 Création d'une variable de type chaîne

Il existe plusieurs possibilités pour créer une variable. On n'en présente que deux, *a priori* les plus utiles :

- `identificateur=mot`

crée la variable `identificateur` avec `mot` comme contenu. **Il ne doit pas y avoir d'espace de chaque côté**

**du signe =.** D'autre part, *mot* est sujet à l'interprétation des caractères spéciaux, y compris les blancs. Si on veut faire figurer ces caractères dans *mot*, il faudra les protéger par des apostrophes (*quotes*), guillemets ou backslash.

 En réalité, ce type de création de variables suit la syntaxe plus complexe suivante :

`{ identificateur=[ mot ] } [ commande ]`

Dans ce cas, on peut créer plusieurs variables (dont certaines peuvent avoir un contenu vide si l'on omet le *mot*). Si une *commande* est présente, alors ces variables ne sont connues que dans le processus exécutant la *commande* et pas pour le bash en cours. Voir les exemples d'erreurs à ne pas commettre pour plus de précisions.


• **let** *identificateur*[ *op* ] = *expression*

si l'on omet *op*, crée la variable *identificateur* en lui affectant le résultat de l'évaluation de *expression* qui doit être une expression arithmétique, dans laquelle peut apparaître des parenthèses ainsi que les opérateurs suivants (dont la quasi totalité existe en C/C++), listés par ordre de priorité décroissante (les opérateurs d'une même cellule ont la même priorité) :

<i>id</i> ++	post-incrémentation d'une variable (même non numérique)
<i>id</i> --	post-décrémentation d'une variable (même non numérique)
++ <i>id</i>	pré-incrémentation d'une variable (même non numérique)
-- <i>id</i>	pré-décrémentation d'une variable (même non numérique)
+	plus unaire
-	moins unaire
!	négation logique (booléenne)
~	complément à 1
**	puissance
*	multiplication
/	division entière
%	modulo (reste de la division entière)
+	addition
-	soustraction
<<	décalage à gauche
>>	décalage à droite
<=	inférieur ou égal
>=	supérieur ou égal
<	inférieur
>	supérieur
==	test d'égalité
!=	différent
&	AND : et bit à bit
^	XOR : ou exclusif bit à bit
	OR : ou bit à bit
&&	et logique (booléen)
	ou logique (booléen)
<i>expr</i> ? <i>expr</i> : <i>expr</i>	opérateur conditionnel

**Il ne doit pas y avoir d'espace de chaque côté du signe =.** *expression* étant sujette à l'interprétation des caractères spéciaux et des blancs, ils doivent être protégés. Les comparaisons et les opérateurs booléens sont évalués à 1 si l'expression est vraie et à 0 sinon.

Enfin, *op* peut être l'un des opérateurs suivants : `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^`, `|`, ce qui est interprété comme en C/C++, mais attention de protéger les (séquences de) caractères spéciaux (`*`, `<<`, `>>`, `&`, `|`).

 En réalité, la syntaxe de **let** est plus exactement :

**let** *expression* { *expression* }

qui évalue dans l'ordre chaque *expression*, contenant éventuellement des opérateurs d'affectation (`=`, `+=`, `*=`, ...) qui sont des opérateurs comme les autres et ont la plus faible priorité.


Pour évaluer une seule *expression*, y compris pour créer une variable, il est certainement plus commode de ne pas utiliser **let** mais plutôt à la place la syntaxe équivalente :

( (*expression*) )

qui est décrite dans la section **16.C.1**. Elle donne exactement le même résultat mais il n'est pas nécessaire de protéger les espaces ni les caractères spéciaux présents dans les opérateurs (`*`, `&`, ...).

### Exemples (bon usage de la première forme)

\$ **ma\_var1=hello**

 création de la variable **ma\_var1** contenant *hello*

\$ **ma\_var1='chaîne de plusieurs mots'**

 modification de **ma\_var1** pour contenir la chaîne *chaîne\_de\_plusieurs\_mots*


\$ **var2=1500**

 création de la variable **var2** contenant **la chaîne** *1500*


□

### Exemples (bon usage de la deuxième forme)


\$ **let var3='2\*\*5'**

 création de **var3** contenant la chaîne *32* (résultat de  $2^5$ )


\$ **let var4='++var2 \* 3'**

 création de **var4** contenant la chaîne *4503* ( $= 1501 \times 3$ ). De plus, la variable **var2** a été incrémentée et vaut maintenant la chaîne *1501*.


\$ **let var5='var2 > 3'**

 création de **var5** qui contient la chaîne *1* (résultat de la comparaison du contenu de **var2** et de 3)

\$ **let x='11 & 7'**

 création de **x** qui contient la chaîne *3* (résultat du ET bit à bit entre 11 et 7)

\$ **let x+=2**

 modification de **x** qui vaut maintenant la chaîne *5*

□

### Exemples (quelques erreurs à ne pas commettre)

```
$ var=deux mots
```

```
-bash: mots: command not found
```

- ⇒ *puisque l'espace séparant `deux` et `mots` n'est pas protégé, bash traite cette ligne selon la syntaxe plus complexe de création de variables : création de `var` valant `deux` qui ne sera connue que dans le processus exécutant la commande `mots` (qui ici n'existe pas, d'où le message d'erreur).*

```
$ let x=var2 > 4
```

- ⇒ *Cette commande ne produit pas d'erreur mais n'a probablement pas l'effet souhaité (placement dans `x` du résultat de la comparaison du contenu de `var2` et de 4). En effet, il n'y a pas de protection du caractère spécial `>` qui est traité comme la redirection de la sortie standard. Ainsi, bash crée la variable `x` contenant 1500 (si `var2` vaut 1500), et un fichier vide nommé 4.*

```
$ let x=11 & 7
```

```
[1] 15197
```

```
-bash: 7: command not found
```

```
[1]+  Done                  let x=11
```

- ⇒ *Puisque le `&` n'est pas protégé, il est traité comme le placement en tâche de fond de la commande qui précède (soit `let x=11`). Au final, `x` est créée dans un nouveau processus (qui se termine aussitôt) et vaut 11. Parallèlement, bash tente d'exécuter la commande 7 qui n'existe pas. À l'issue de cette ligne de commandes, la valeur de `x` n'aura pas changé. On reviendra sur la portée des variables plus loin.*

```
$ let x&=7
```

```
[1] 20778
```

```
-bash: =7: command not found
```

```
[1]+  Done                  let x
```

- ⇒ *Ici, on veut utiliser la contraction de `let x='x & 7'` en utilisant `let x&=7`. Mais puisque `&` n'est pas protégé, il s'ensuit une erreur similaire à la précédente. Il aurait fallu écrire `let x\&=7` ou encore `let 'x&=7'`, etc.*



**Ces exemples montrent que dans la plupart des cas, il vaut mieux encadrer ce qui suit le signe = par des quotes ou des guillemets. La différence entre ces deux types de protections sera clarifiée dans la section 15.A.3.**

□

## 15.A.2 unset : supprimer une variable ou fonction

Pour supprimer une variable, il faut utiliser la commande interne **unset**.

### Synopsis

```
unset [-v | -f] nom {nom}
```

L'option **-v** demande de ne supprimer que les variables *nom*. L'option **-f** demande de ne supprimer que les fonctions *nom*. Sans option, bash cherchera d'abord à supprimer la variable *nom* et si elle n'existe pas, tentera de supprimer la fonction *nom*. La tentative de suppression d'une variable ou fonction qui n'existe pas ne produit pas d'erreur.



### 15.A.3 Substitution des variables


Pour obtenir le contenu d'une variable, il faut faire précéder son identificateur du caractère spécial **\$** (qui a d'autres fonctions comme nous le verrons plus tard). En effet, lorsque le **\$** n'est pas protégé, alors les écritures

`$identificateur`


ou

`${identificateur}`

s'appellent une **substitution de la variable** *identificateur* et sont remplacées par le contenu de la variable *identificateur*, avant que la commande dans laquelle elles apparaissent ne soit exécutée. L'utilisation des accolades sera discutée à la fin de cette section.

 Il y a deux façons de protéger le **\$** : entre quotes ou en le faisant précéder d'un backslash. Il n'est donc pas protégé entre guillemets (sauf s'il est précédé d'un backslash).

Si une substitution pour une variable inexistante est demandée (par exemple `$var`, où `var` n'existe pas), bash ne produit pas d'erreur et remplace la substitution par une chaîne vide.

 On peut demander à bash de produire une erreur en cas de substitution de variable inexistante en activant l'option *nounset* par la commande `shopt -os nounset` (on peut la désactiver ensuite par `shopt -ou nounset`).


#### Exemples

Reprenons les bons exemples de la section précédente pour faire afficher les résultats avec la commande **echo** :

```
$ ma_var1=hello
$ echo $ma_var1
hello
```

 la substitution a lieu **avant** l'exécution de la commande : la commande réellement exécutée est donc `echo hello`

```
$ ma_var1='chaîne de plusieurs mots'
$ echo $ma_var1
chaîne de plusieurs mots
$ var2=1500
$ echo $var2
1500
$ let var3='2**5'
$ echo $var3
32
$ let var4='++var2 * 3'
$ echo $var4
4503
$ echo $var2
1501
```

 on remarque que pour **let** (et de façon générale partout où bash attend une expression arithmétique), il n'est pas nécessaire de faire précéder les identificateurs par **\$**. Ce n'est toutefois pas interdit.

```
$ let var5='var2 > 3'
$ echo $var5
1
$ let x='11 & 7'
$ echo $x
3
$ let x+=2
$ echo $x
5
```



### Exemples (un peu plus "fouillés")

```
$ var1="une variable"
$ var2="construite avec $var1"
```

⇒ la commande réellement exécutée est **var2="construite avec une variable"**

```
$ echo $var2
construite avec une variable
$ var3="construite avec \$var1"
```

⇒ cette fois, le **\$** est protégé par le backslash et la substitution n'a pas lieu

```
$ echo $var3
construite avec $var1
```

⇒ la substitution a lieu pour **\$var3** mais pas sur son contenu (et notamment **\$var1**)

```
$ var3='construite avec $var1'
```

⇒ autre protection de **\$** mais par les quotes

```
$ echo $var3
construite avec $var1
$ var1="des____espaces"
$ echo $var1
des espaces
```

⇒ la ligne de commandes devient **echo\_des\_\_\_\_espaces**. Donc **echo** écrit ses 2 arguments en les séparant par un seul espace.

```
$ echo $var2
construite avec une variable
```

⇒ la modification de **var1** n'a pas modifié **var2**

```
$ echo "$var1"
des____espaces
```

⇒ les guillemets font que **echo** ne reçoit qu'un seul argument (qui contient les espaces)

```
$ echo "cette $variable n'existe pas"
cette n'existe pas
```

⇒ **\$variable** n'existant pas, sa substitution ne donne rien (chaîne vide) sans provoquer d'erreur

```
$ shopt -os nounset
$ echo "cette $variable n'existe pas"
-bash: variable: unbound variable
```

⇒ il y a erreur si l'option **nounset** est activée

□

## Utilité des accolades

Nous avons déjà rencontré ces accolades lors de la présentation des paramètres positionnels des fonctions (voir chapitre 9), car elles sont nécessaires pour accéder aux paramètres de numéro supérieur à 9 (par exemple, **\${10}**).

Une première utilisation est de clairement indiquer l'identificateur de la variable sur laquelle la substitution doit être opérée. En effet, supposons que la variable **var** soit créée et contienne **-bidule-**, alors la commande suivante :

```
x="truc$varchouette"
```

crée la variable **x** contenant seulement **truc** si la variable **varchouette** n'existe pas, alors que la commande :

```
x="truc${var}chouette"
```

crée la variable **x** contenant **truc-bidule-chouette**.

De plus, les accolades permettent un grand nombre de manipulations sur (le contenu de) la variable, qui sont présentées ci-après, ainsi que sur les tableaux (voir section 15.B page 217)..

## 15.A.4 Substitutions étendues des variables

Toutes ces substitutions nécessitent l'emploi des accolades. Elles apportent de nombreuses fonctionnalités dont certaines seront particulièrement utiles dans les scripts bash.


### 15.A.4.a Substitutions selon l'existence et le contenu

- **\${identificateur:-chaîne}** : utilisation d'une valeur par défaut  
Si la variable *identificateur* existe et n'est pas vide, remplacer par son contenu, sinon par *chaîne*.
- **\${identificateur:=chaîne}** : attribuer une valeur par défaut  
Si *identificateur* existe et n'est pas vide, remplacer par son contenu, sinon lui attribuer<sup>1</sup> *chaîne* avant de remplacer par son contenu.
- **\${identificateur:? [chaîne]}** : afficher une erreur si indéfinie ou nulle  
Si *identificateur* existe et n'est pas vide, remplacer par son contenu, sinon afficher le message indiqué par *chaîne* sur la sortie d'erreur et terminer le shell s'il n'est pas interactif (cas des scripts bash). Si *chaîne* n'est pas présente, le message affiché est le message par défaut de bash.

1. On ne peut pas modifier les paramètres positionnels ni les paramètres spéciaux de cette façon.

- `${identificateur:+chaîne}` : utilisation d'une valeur alternative  
Si la variable *identificateur* existe et n'est pas vide, remplacer par *chaîne*, sinon remplacer par la chaîne nulle.

Pour ces substitutions, le `:` est optionnel. S'il n'est pas présent, alors bash ne teste que l'existence de la variable (il ne teste pas si elle est vide).

 *chaîne* peut contenir des blancs mais est sujette à la substitution du tilde, des variables, des commandes (voir plus loin), et des expressions arithmétiques (voir plus loin). On peut protéger les caractères spéciaux qui provoquent ces substitutions comme d'habitude (mais ici à l'intérieur des accolades).

## Exemples

Pour illustrer ces substitutions, on va prendre 3 variables :


- **nonvide** qui n'est pas vide
- **vide** qui est vide
- **aucune** qui n'existe pas

```
$ nonvide="non vide"
$ vide=""
```

Utilisation d'une valeur par défaut :

```
$ echo "${nonvide:-est inconnue ou vide}"
non vide
$ echo "${vide:-est inconnue ou vide}"
est inconnue ou vide
$ echo "${vide-est inconnue}"
```

```
$ echo "${aucune-est inconnue}"
est inconnue
```

 on remarque que le test est différent en l'absence du :

Attribuer une valeur par défaut :

```
$ echo "${nonvide:=valeur si vide ou inconnue}"
non vide
$ echo "$nonvide"
non vide
$ echo "${vide:=valeur si vide ou inconnue}"
valeur si vide ou inconnue
$ echo "$vide"
valeur si vide ou inconnue
$ vide=""
$ echo "${vide=valeur si inconnue}"

$ echo "$vide"
```

```
$ echo "${aucune=valeur si inconnue}"
valeur si inconnue
$ echo "$aucune"
valeur si inconnue
```

Afficher une erreur si indéfinie ou nulle :

```
$ unset aucune
```

⇒ d'abord, on supprime la variable **aucune** qui a été créée précédemment

```
$ echo "${nonvide:?cette variable est vide ou inconnue}"
```

```
non vide
```

```
$ echo "${vide:?cette variable est vide ou inconnue}"
```

```
-bash: vide: cette variable est vide ou inconnue
```

```
$ echo "${vide?cette variable est inconnue}"
```

```
$ echo "${aucune?cette variable est inconnue}"
```

```
-bash: aucune: cette variable est inconnue
```

Valeur alternative :

```
$ echo "${nonvide:+existe et non vide}"
```

```
existe et non vide
```

```
$ echo "${vide:+existe et non vide}"
```

```
$ echo "${vide+existe}"
```

```
existe
```

```
$ echo "${aucune+existe}"
```

□

#### 15.A.4.b Substitutions de la longueur et d'une sous-chaîne

- `${#identificateur}` : obtenir la longueur du contenu  
Est remplacé par le nombre de caractères du contenu de la variable *identificateur*
- `${identificateur:décalage[:longueur]}`  
Est remplacé par la sous-chaîne du contenu de la variable *identificateur*, commençant à *décalage* et de *longueur* caractères. Le premier caractère porte le numéro 0. *décalage* et *longueur* sont des expressions arithmétiques (comme dans **let**). *longueur* est optionnelle : si elle n'est pas indiquée, alors la sous-chaîne sera de *décalage* jusqu'à la fin. Si elle est précisée elle doit être positive. *décalage* peut être négatif, dans ce cas le décalage sera pris à partir de la fin (le dernier caractère porte le numéro -1). Cependant, si *décalage* commence par le signe -, il faut le séparer du : par un espace pour ne pas être confondu avec l'attribution d'une valeur par défaut.

**i** Cette substitution a d'autres fonctionnalités qui seront présentées avec les tableaux et les paramètres positionnels.

#### Exemples

```
$ var=abcdefghijklmnopqrstuvwxy
```

```
$ echo ${#var}
```

```
26
```

⇒ **var** contient 26 caractères

```
$ echo ${var:12}
```

```
mnopqrstuvwxyz
```

⇨ affichage du caractère 12 jusqu'à la fin

```
$ echo ${var:12:5}
```

```
mnopq
```

⇨ affichage de 5 caractères à partir du 12<sup>e</sup>

```
$ echo ${var:_-1:1}
```

```
z
```

⇨ affichage du dernier caractère de **var**

```
$ echo ${var:0:${#var}/2}
```

```
abcdefghijklm
```

⇨ affichage de la moitié des caractères de **var** depuis le début

□

#### 15.A.4.c Substitutions de sous-chaînes avec les motifs

Dans ce qui suit, *motif* est une chaîne de caractères construite avec les mêmes caractères (sauf les accolades) que pour les motifs de noms de fichiers et ont la même signification : **\***, **?**, **[]**.

- **`${identificateur#motif}`**  
Si *motif* correspond **au début** du contenu de *identificateur*, alors retourne le contenu privé de la plus **petite** chaîne, située au début, à laquelle le *motif* correspond. Sinon, retourne le contenu de *identificateur*.
- **`${identificateur##motif}`**  
Si *motif* correspond au début du contenu de *identificateur*, alors retourne le contenu privé de la plus **grande** chaîne, située au début, à laquelle le *motif* correspond. Sinon, retourne le contenu de *identificateur*.
- **`${identificateur%motif}`**  
Si *motif* correspond **à la fin** du contenu de *identificateur*, alors retourne le contenu privé de la plus **petite** chaîne, située à la fin, à laquelle le *motif* correspond. Sinon, retourne le contenu de *identificateur*.
- **`${identificateur%%motif}`**  
Si *motif* correspond **à la fin** du contenu de *identificateur*, alors retourne le contenu privé de la plus **grande** chaîne, située à la fin, à laquelle le *motif* correspond. Sinon, retourne le contenu de *identificateur*.
- **`${identificateur/motif/remplacement}`**  
Remplace par *remplacement* la plus longue partie du contenu de *identificateur* à laquelle le *motif* correspond. Seule la première correspondance est remplacée. Si *motif* commence par **#**, la chaîne correspondante doit être située au début. S'il commence par **%**, elle doit être située à la fin.
- **`${identificateur//motif/remplacement}`**  
Remplace par *remplacement* les plus longues parties du contenu de *identificateur* auxquelles le *motif* correspond. Toutes les correspondances sont remplacées.

❗ Ces substitutions ont d'autres fonctionnalités qui seront présentées avec les tableaux et les paramètres positionnels.

✍ Comment se souvenir de la fonction des caractères # et % ? L'un indique le début et l'autre la fin. Simplement, # est le dièse (qui commence par d), c'est le début. L'autre marque donc la fin.

## Exemples

*Les quatre premières substitutions sont particulièrement utiles pour effectuer des manipulations basiques sur des références de fichiers, les deux dernières un peu moins.*

```
$ ref=/chemin/vers/un/vieux/fichier.txt.vieux
$ echo ${ref#*/}
vers/un/vieux/fichier.txt.vieux
$ echo ${ref##*/}
fichier.txt.vieux
```

⇒ on ne garde que le nom du fichier

```
$ echo ${ref##*.}
vieux
```

⇒ on ne garde que l'extension du fichier

```
$ echo ${ref%.*}
/chemin/vers/un/vieux/fichier.txt
```

⇒ on supprime l'extension du fichier (ici .vieux)

```
$ echo ${ref%/*}
/chemin/vers/un/vieux
```

⇒ on ne garde que le chemin vers le fichier

```
$ echo ${ref%%.*}
/chemin/vers/un/vieux/fichier
$ echo ${ref/vieux/old}
/chemin/vers/un/old/fichier.txt.vieux
$ echo ${ref/%vieux/old}
/chemin/vers/un/vieux/fichier.txt.old
$ echo ${ref//vieux/old}
/chemin/vers/un/old/fichier.txt.old
```

□

## 15.B Les tableaux

bash permet la création de tableaux à une seule dimension. Un tableau a un identificateur et une certaine taille qui est déterminée automatiquement lors de sa création et qui peut évoluer. Chaque élément du tableau est une variable à laquelle on accède en indiquant le tableau. Le premier élément d'un tableau se trouve à l'indice 0.

❗ En réalité, bash considère qu'une variable n'est autre qu'un tableau à un seul élément.

## 15.B.1 Création d'un tableau

La manière la plus simple pour créer un tableau est la suivante :

```
identificateur=( { mot } )
```

où il ne doit pas y avoir de blanc avant le = (.

Cette instruction crée<sup>2</sup> le tableau *identificateur*, d'une taille égale au nombre de *mots*, et affecte à chaque élément le *mot* correspondant.

### Exemple

```
$ tab=(zéro un deux trois)
```

➡ crée le tableau **tab** de 4 éléments où l'élément 0 contient *zéro* et l'élément 3 contient *trois*. □

On peut aussi créer un tableau en affectant directement les éléments souhaités, en indiquant le tableau en utilisant la forme *identificateur* [*indice*], où *indice* est une **expression arithmétique** (comme dans **let**). Ainsi, pour créer le même tableau que précédemment, on peut écrire :

```
$ tab[0]=zéro
$ tab[1]=un
$ tab[2]=deux
$ tab[3]=trois
```

✍ Il en découle que le tableau "grossit" au fur et à mesure qu'on lui ajoute des éléments.

Une autre écriture équivalente aurait été :

```
$ tab=([1]=un [2]=deux [0]=zéro [3]=trois)
```

et ce, quel que soit l'ordre d'affectation des éléments.

Par ailleurs, **bash** accepte la création de tableaux à trous ! Notamment, on aurait pu écrire :

```
$ tab=([0]=zéro [3]=trois)
```

puis compléter le tableau par :

```
$ tab[1]=un
$ tab[2]=deux
```



**Attention cependant à ne pas le compléter avec l'instruction utilisant les parenthèses :**

```
$ tab=([1]=un [2]=deux)
```

**car cela écraserait le tableau.**

2. Si une variable ou un tableau *identificateur* existait déjà, elle/il est écrasé(e).




## 15.B.2 Suppression d'un tableau ou d'un élément

Pour supprimer un tableau, on utilise aussi la commande **unset**. On peut aussi supprimer un élément particulier d'un tableau en écrivant :

```
unset identificateur [indice]
```

## 15.B.3 Substitutions propres aux tableaux

Pour obtenir le contenu d'un élément d'un tableau il faut écrire `${identificateur [indice] }`, les **accolades étant obligatoires**.

 Toutes les substitutions étendues des variables s'appliquent aussi aux éléments des tableaux. Il faut simplement remplacer dans leur définition *identificateur* par *identificateur* [*indice*] . D'autre part, ces substitutions peuvent s'appliquer à tous les éléments d'un tableau, comme nous le verrons à la fin de cette section.

D'un autre côté, les tableaux admettent quelques substitutions qui leur sont propres.

### 15.B.3.a Obtenir tous les éléments d'un tableau

Il existe deux possibilités :

- `${identificateur [@] }`
- `${identificateur [*] }`

Si elles ne sont pas utilisées entre **guillemets**, elles sont toutes les deux remplacées par le contenu des éléments non vides du tableau, séparés par un espace. Si elles sont placées entre guillemets, tous les éléments initialisés (même vides) du tableau sont utilisés et leur résultat est différent.

Supposons qu'un tableau **tab** ait été créé ainsi :

```
$ tab=(zéro un deux)
```

alors, `"${tab[@] }"` sera remplacé par :

```
"zéro" _ "un" _ "deux"
```

soit 3 chaînes séparées par un espace, alors que `"${tab[*] }"` sera remplacée par :

```
"zéroCunCdeux"
```


soit **une seule chaîne** dans laquelle les éléments sont séparés par un caractère *c*, qui est le premier caractère de la variable **IFS** (pour *Internal Field Separators*). Cette variable contient par défaut, dans l'ordre, les 3 caractères espace, tabulation et retour à la ligne.

Ainsi, par défaut on obtiendra pour `"${tab[*]}"` :

```
"zéro_un_deux"
```

Mais on peut modifier **IFS** comme avec les commandes suivantes :

```
$ IFS=' , '
$ echo "${tab[*]}"
zéro,un,deux
```

 Cette différence peut paraître anodine mais en réalité elle est très importante. D'une part, l'utilisation de **IFS** amène une souplesse assez puissante qu'on retrouvera à plusieurs reprises dans d'autres cas. D'autre part, la première forme permet de copier le contenu d'un tableau, mais pas la seconde comme le montre l'exemple suivant.

### Exemple

```
$ tab=(un tableau 'de quatre' éléments)
```

⇨ création d'un tableau de 4 éléments : le troisième valant `de_quatre`

```
$ tab1=("${tab[@]}")
```

⇨ copie incorrecte du tableau car cela crée un tableau **tab1** de 5 éléments.

```
$ tab1=("${tab[*]}")
```

⇨ même effet


```
$ tab1=("${tab[@]}")
```

⇨ copie correcte du tableau dans le tableau **tab1**

```
$ tab2=("${tab[*]}")
```

⇨ création d'un tableau d'un seul élément

□

 Entre guillemets, tous les éléments initialisés, même vides, sont utilisés comme le montre l'exemple suivant.

### Exemple

```
$ tab=(zéro '' deux [4]=quatre)
```

⇨ création d'un tableau à trous : l'élément 1 est vide mais existe, alors que l'élément 3 n'existe pas (il n'a pas été initialisé)

```
$ echo "${tab[@]}"
zéro_ _deux_ quatre
```

⇒ c'est comme si l'on avait tapé la commande : **echo "zéro" "" "deux" "quatre"** : les 2 espaces entre *zéro* et *deux* séparent en fait la chaîne vide qui constitue l'élément 1, alors que l'élément 3 qui n'existe pas n'est pas écrit. Au final, **echo** a bien reçu 4 arguments qu'il a écrit en les séparant par un espace.

```
$ IFS=' , '
$ echo "${tab[*]}"
zéro,,deux, quatre
```

⇒ c'est comme si l'on avait tapé la commande : **echo "zéro,,deux, quatre"** : l'élément 1 (vide) est écrit entre les 2 virgules séparant *zéro* et *deux*, alors que l'élément 3 n'est pas écrit. Ici, **echo** ne reçoit qu'un argument.

□

### 15.B.3.b Obtenir la liste des indices des éléments d'un tableau

Il existe deux possibilités :

- `${!identificateur[@]}`
- `${!identificateur[*]}`

Si elles ne sont pas utilisées entre **guillemets**, elles sont toutes les deux remplacées par les indices des éléments créés du tableau (même vides), séparés par un espace. Si elles sont placées entre guillemets, alors leur résultat est différent, de la même manière qu'il l'est pour l'obtention de tous les éléments du tableau.

#### Exemple

```
$ tab=([0]=zéro [1]=" " [4]=quatre [10]=dix)
$ IFS=' , '
$ echo "${!tab[@]}"
0 1 4 10
```

⇒ après remplacement, la ligne de commandes devient : **echo "0" "1" "4" "10"**

```
$ echo "${!tab[*]}"
0,1,4,10
```

⇒ après remplacement, la ligne de commandes devient : **echo "0,1,4,10"**

□

### 15.B.3.c Obtenir le nombre d'éléments d'un tableau

Il faut utiliser `${#identificateur[*]}` ou `${#identificateur[@]}`. Ces deux formes sont équivalentes, même entre guillemets. On obtient alors le nombre d'éléments du tableau qui ont été créés, même vides.

#### Exemple (suite de l'exemple précédent)

```
$ echo ${#tab[@]}
4
$ echo ${#tab[*]}
4
```

□

### 15.B.3.d Application des substitutions de sous-chaînes à l'ensemble d'un tableau

Toutes les substitutions de sous-chaînes peuvent s'appliquer à un tableau entier. Plutôt que *identificateur* dans leur définition il faut écrire *identificateur*[@] ou *identificateur*[\*]. La différence entre les deux écritures devrait être claire maintenant.

Les substitutions de sous-chaînes avec les motifs (voir section 15.A.4.c) pour un tableau s'appliquent à tous les éléments du tableau et ne méritent pas d'explications supplémentaires.

En revanche, la substitution d'une sous-chaîne (voir section 15.A.4.b) appliquée à un tableau est un cas particulier. En effet, le résultat des substitutions suivantes :

```
${identificateur[@]:décalage[:longueur]}
```

et

```
${identificateur[*]:décalage[:longueur]}
```

est d'obtenir la partie du tableau *identificateur* commençant au *décalage*<sup>ième</sup> élément initialisé et de *longueur* éléments initialisés. *longueur* peut être omis, ce qui revient à tous les éléments à partir du *décalage*<sup>ième</sup>. @ et \* ont toujours la signification qu'on leur connaît. *décalage* peut être négatif pour partir de la fin du tableau (le dernier élément portant le numéro -1) mais dans ce cas les éléments non initialisés comptent pour la numérotation de *décalage* (voir exemple).

### Exemples

```
$ tab=(/chemin0/fic0.txt '' /chemin2/fic2.txt [10]=/chemin10/fic10.txt)
$ IFS=' , '
```

#### Substitutions avec motifs sur l'ensemble des éléments

```
$ echo "${tab[@]#*/}"
fic0.txt fic2.txt fic10.txt
$ echo "${tab[*]%.*}"
/chemin0/fic0,,/chemin2/fic2,/chemin10/fic10
$ echo "${tab[@]/chemin/vers}"
/vers0/fic0.txt /vers2/fic2.txt /vers10/fic10.txt
$ echo "${tab[@]//i/xxx}"
/chemXXXn0/fXXXc0.txt /chemXXXn2/fXXXc2.txt /chemXXXn10/fXXXc10.txt
```

#### Parties du tableau

```
$ echo "${tab[*]:0:3}"
/chemin0/fic0.txt,,/chemin2/fic2.txt
```

⇨ 3 éléments à partir du premier

```
$ echo "${tab[*]:2:2}"
/chemin2/fic2.txt,/chemin10/fic10.txt
```

⇨ 2 éléments initialisés à partir du 3<sup>e</sup> élément initialisé (on obtient les éléments d'indices 2 et 10)

```
$ echo "${tab[*]:3:1}"
/chemin10/fic10.txt
```

⇒ un élément initialisé à partir du 4<sup>e</sup> élément initialisé (on obtient l'élément d'indice 10)

```
$ echo "${tab[@]:-2:2}"
/chemin10/fic10.txt
```

⇒ les choses se compliquent un peu (☹ et ☹) : on pourrait penser qu'on obtiendrait les 2 derniers éléments. Dans le cas d'un décalage négatif, tous les indices comptent même les éléments non initialisés. Le dernier élément (d'indice 10) est représenté par le décalage -1, et l'avant dernier élément initialisé (d'indice 2) est représenté par le décalage -9 ! Je suis d'accord, ce n'est pas super. . .

```
$ echo "${tab[@]:-9:2}"
/chemin2/fic2.txt /chemin10/fic10.txt
```

□

✍ Ce dernier exemple montre que de travailler avec des tableaux à trous est rarement une bonne idée. Dans le cas où cela est nécessaire, il sera certainement pertinent de travailler à partir des indices des éléments initialisés (voir section 15.B.3.b).

## 15.C Afficher les variables, tableaux et fonctions existants

La commande interne **declare** permet de déclarer (créer) des variables, tableaux et fonctions mais aussi d'afficher ceux existants.

### Synopsis

**declare** [-afFx] { *identificateur* }

Sans option ni argument, **declare** affiche la liste de toutes les variables, tableaux et fonctions existants.

Les options demandent de restreindre l'affichage à certaines catégories (variables, tableaux ou fonctions) et affiche la déclaration et la valeur de chaque élément de la catégorie sélectionnée.

- L'option **-a** demande l'affichage des tableaux uniquement. En utilisant l'option **-p**, cet affichage est restreint aux *identificateurs* indiqués ;
- L'option **-f** demande l'affichage des fonctions uniquement ainsi que leur corps. Si des *identificateurs* sont précisés, l'affichage est restreint à ces fonctions, mais il ne faut pas utiliser l'option **-p**.
- L'option **-F** demande l'affichage des noms des fonctions uniquement. Si des *identificateurs* sont précisés, l'affichage est restreint à ces fonctions, mais il ne faut pas utiliser l'option **-p**.
- L'option **-x** demande de restreindre cet affichage aux variables, tableaux et fonctions d'environnement (voir section 17.D page 273). Elle peut être combinée avec l'option **-f** ou **-F** mais on ne peut alors préciser d'*identificateur*. Elle peut être aussi combinée avec **-a**<sup>3</sup> et/ou **-p**.



**L'option -p ne doit être utilisée que si les identificateurs sont des variables ou des tableaux.**

3. Actuellement, combiner **-x** et **-a** ne donnera rien car les tableaux d'environnement ne sont pas encore implémentés.

## 15.D Variables et tableaux internes de bash

Bash crée lui-même un certain nombre de variables et de tableaux dont certains ne sont pas modifiables et d'autres sont mis à jour par bash. Certaines variables et tableaux ont aussi un statut particulier : celui d'être d'environnement (voir section [17.D page 273](#)). D'autres variables, s'il elles sont créées, modifient le comportement de bash. Seule une infime partie des variables et tableaux internes de bash est décrite ci-dessous (se reporter au manuel en ligne pour les avoir toutes) :

- **\$** : (on ne peut qu'utiliser sa substitution **\$\$**) PID du processus courant ;
- **PPID** : PID du processus père ;
- **PS1** : utilisée uniquement par les bash interactifs, contient une chaîne indiquant quelle doit être l'invite de commandes (le *prompt*) à afficher. Elle est personnalisable et contient souvent par défaut la chaîne "**\u@\h:\w\\$\_**", où :
  - ◊ **\u** est remplacé par le nom d'utilisateur
  - ◊ **\h** est remplacé par le nom (court) de la machine
  - ◊ **\w** est remplacé par le répertoire de travail, où le répertoire d'accueil est éventuellement remplacé par **~**
  - ◊ **\\$\_** est remplacé par **#** si l'on est root, et par **\$** sinon
- **PS2** : le prompt de niveau 2 qui vaut par défaut "**>\_**". Personnalisable aussi, il est affiché par un bash interactif lorsque l'utilisateur a tapé sur Entrée mais n'a pas terminé sa commande, par exemple est en train de définir une fonction, écrire une boucle, etc. Il sera affiché jusqu'à ce que l'utilisateur tape le mot-clé ou le symbole de fermeture de la commande ;
- **PS3** : l'invite que bash affiche lorsque l'instruction **select** est exécutée (voir section [16.D.7 page 257](#)) : vaut par défaut "**#?\_**" ;
- **PS4** : chaîne affichée lors de l'exécution d'un script lorsque l'option **-x** a été passée à bash. Par défaut, vaut "**+\_**" ;
- **IFS** : (*Internal Field Separator*) contient les caractères de séparation des mots utilisés pour la décomposition des mots (voir section [18.D page 278](#)) et pour la commande interne **read** (voir section [16.E.2 page 260](#)). Le premier caractère de **IFS** est aussi utilisé pour la substitution des tableaux ou paramètres positionnels indicés par **\*** et placés entre guillemets ;
- **BASH** : contient le chemin complet de l'interpréteur bash (le plus souvent **/bin/bash**) ;
- **REPLY** : variable utilisée par l'instruction **select** (voir section [16.D.7 page 257](#)) et la commande **read** (voir section [16.E.2 page 260](#)) ;
- **RANDOM** : nombre aléatoire entre 0 et  $2^{15} - 1$  ;
- **UID** : numéro de l'utilisateur réel. Cette variable est créée lors de l'ouverture de la session utilisateur et initialisée avec le champ *uid* de la ligne concernant l'utilisateur dans le fichier **/etc/passwd** ;
- **GROUPS** : tableau contenant les numéros des groupes auxquels l'utilisateur appartient ;
- **EUID** : numéro de l'utilisateur effectif.

Variables d'environnement :

- **HOME** : répertoire personnel de l'utilisateur. Cette variable est créée lors de l'ouverture de la session utilisateur et initialisée avec le champ *répertoire* de la ligne concernant l'utilisateur dans le fichier **/etc/passwd**. Elle est utilisée par la commande interne **cd** quand elle n'a pas d'argument et pour l'expansion du tilde ;
- **USER** : nom de l'utilisateur ;
- **PWD** : chemin absolu du répertoire de travail. Cette variable détermine notamment ce qu'écrit le prompt lorsque **PS1** contient **\w** ;

- **OLDPWD** : chemin absolu du répertoire de travail précédent. Mise à jour automatiquement par bash lorsque le répertoire de travail change avec **cd**. C'est aussi le répertoire destination de **cd** quand son argument est **-** ;
- **PATH** : liste de répertoire séparés par **:** dans lesquels bash recherche les commandes externes ;
- **BASH\_ENV** : cette variable est utilisée lorsqu'un sous-shell est créé pour exécuter un script. Si elle existe et est d'environnement, elle doit contenir le chemin d'un fichier que doit exécuter ce sous-shell (par **source**) avant d'exécuter le script. Selon la distribution (comme une Debian), cette variable n'est pas créée par défaut. Sur d'autres, comme la Mandriva, elle contient **\$HOME/.bashrc** pour qu'un shell non interactif exécute ce fichier de l'utilisateur.





# Chapitre 16

## Les scripts et la programmation bash

---

Bash n'est pas qu'un simple interpréteur de commandes interactif. C'est aussi un langage de programmation qui permet l'écriture de boucles et d'instructions conditionnelles. Bien qu'il soit possible (et parfois très pratique) d'utiliser ces instructions en ligne de commandes, elles sont davantage destinées à être utilisées dans des programmes, appelés des **scripts shell**, et en particulier des **scripts bash**.

Les scripts bash sont de simples fichiers texte pouvant contenir des suites de commandes que nous avons vues jusqu'à présent : des commandes simples ou complexes (avec des pipes, des redirections, etc.), des déclarations et appels de fonctions, des alias, des variables et tableaux, des boucles, etc. Ils sont interprétés par bash comme si les instructions qu'ils contiennent étaient tapées sur la ligne de commande, et ne nécessitent pas de phase de compilation. Une légère différence est que le bash exécutant un script n'est pas interactif, notamment il n'affiche pas de prompt entre deux instructions du script.

Nous avons déjà rencontré des scripts bash au chapitre 9 : les fichiers `/etc/profile`, `~/.bash_profile`, `/etc/bash.bashrc` et `~/.bashrc`. Ces fichiers sont notamment exécutés par un (login) shell bash pour personnaliser l'environnement de l'utilisateur.

Les scripts que nous écrirons auront généralement une autre fonction : celle d'exécuter un certain nombre de tâches, comme pourrait le faire un utilitaire quelconque.

### 16.A Les scripts bash

#### 16.A.1 Exécution d'un script

Soit un fichier `monscript` contenant le texte suivant :

```
cd /tmp
echo    "hello !"
echo -n "je me trouve dans le répertoire : "
pwd
```

alors, il existe au moins trois manières d'exécuter `monscript` :

1. exécuter un autre shell (un sous-shell) pour exécuter le script en tapant :

**bash [options-bash] monscript**

où *options-bash* sont des options passées à bash. La plus utile dans notre cas est l'option **-x** pour que chaque ligne qui va être exécutée soit d'abord affichée (précédée du contenu de la variable **PS4**, qui contient par défaut la chaîne "+\_"). Cela permet de "tracer" un script qui comporte une erreur.

2. rendre le fichier exécutable et l'exécuter en tapant les commandes :

```
chmod +x monscript
./monscript
```

3. demander au shell courant d'exécuter lui-même les commandes qu'il contient en utilisant la commande interne **source**, en tapant :

```
source monscript
ou
. monscript
```

ces deux écritures étant équivalentes.

Dans les trois cas, on obtient le même affichage :

```
$ bash monscript
hello !
je me trouve dans le répertoire : /tmp
$ chmod +x monscript
$ ./monscript
hello !
je me trouve dans le répertoire : /tmp
$ source monscript
hello !
je me trouve dans le répertoire : /tmp
```

**Cependant, ces trois exécutions ne sont pas équivalentes.** En effet, dans la dernière exécution, c'est le shell courant qui exécute les commandes du script. Or celui-ci contient l'instruction **cd** qui le fait changer de répertoire.


Dans la première exécution, le shell courant ne change pas de répertoire : c'est un sous-shell qui a exécuté le script et qui a exécuté **cd**, et c'est donc ce sous-shell qui a changé de répertoire.

Pour la seconde exécution, le shell courant ne change pas de répertoire : c'est aussi un sous-shell qui a exécuté le script, ce qui revient au même que la première exécution.

On peut vérifier tout ça en regardant quel est le répertoire de travail à chaque fois :

```
$ pwd
/home/cyril/essais
$ bash monscript
hello !
je me trouve dans le répertoire : /tmp
$ pwd
/home/cyril/essais
$ ./monscript
hello !
je me trouve dans le répertoire : /tmp
$ pwd
/home/cyril/essais
$ source monscript
hello !
je me trouve dans le répertoire : /tmp
```

```
$ pwd  
/tmp
```

 L'exécution d'un script par **source** n'a de sens que si l'on veut modifier l'environnement du shell courant (changement de répertoire, création de variables/tableaux, de fonctions ou d'alias). Lorsqu'on modifie l'un des fichiers de personnalisation comme `~/.bash_profile` ou `~/.bashrc`, et que l'on veut que le shell courant prenne en compte ces modifications, alors **il faut** utiliser **source**.

## 16.A.2 La ligne shebang

La différence entre la première et la seconde forme d'exécution se situe dans l'interpréteur utilisé. Dans la première forme, qui est plus lourde à écrire, c'est bien entendu bash. Dans la seconde, cela dépend du shell courant et de la première ligne du script. Dans notre cas, le shell courant est bash et la première ligne n'est pas particulière, donc ce sera aussi bash qui exécutera le script, mais on ne peut pas préciser d'option pour bash (notamment l'option **-x**).

C'est là qu'intervient la ligne **shebang** (prononcer « chibang »). Elle est optionnelle et, si présente, **doit être la première ligne du script**. Elle a la forme suivante :


```
#!chemin-interpréteur [options-interpréteur]
```

Tous les shells (sh, bash, csh, tsh, ksh, etc.) interprètent cette ligne de la même manière : elle indique au shell quelle est la commande qu'il doit exécuter pour traiter (exécuter) le script. Si `monscript` contient cette première ligne et que l'on tape :

```
$ ./monscript
```

alors n'importe quel shell lira la ligne *shebang* et exécutera en réalité la commande suivante :

```
chemin-interpréteur [options-interpréteur] ./monscript
```

 Il est conseillé de commencer tous vos scripts bash par la ligne *shebang* suivante :

```
#!/bin/bash [options-bash]
```

## 16.A.3 Les commentaires

Pour tous les shells, le caractère **#** est un caractère spécial qui marque le début d'un commentaire. Tout ce qui se trouve sur une ligne après un **#** non protégé (par des guillemets, quotes ou backslash) est ignoré. La seule exception concerne la ligne *shebang* qui n'est utilisée que si le script est exécuté directement par la deuxième forme d'exécution.

C'est une raison supplémentaire pour placer la ligne *shebang* en début de tous les scripts, et de ne pas hésiter à commenter les scripts. Ainsi, si notre script `monscript` devient :

```
#!/bin/bash

#####
# Nom : monscript
#
# script illustrant quelques concepts sur les scripts bash
#
# Usage : monscript
#
# Auteur : C. Pain-Barre
# Date : 26/02/2008
#####

cd /tmp                                # se place dans /tmp
echo "hello !"                         # dit bonjour
echo -n "je me trouve dans le répertoire : " # commence un message
pwd                                    # affiche le répertoire courant
```


alors, on n'a rien changé au code effectif mais en l'éditant on comprend mieux à quoi il sert...

### 16.A.4 Paramètres positionnels et paramètres spéciaux

Comme les fonctions, les scripts peuvent avoir des arguments (paramètres). On complète alors les trois formes d'exécution ainsi :

```
bash [options-bash] script {argument}
script {argument}
source script {argument}
```

où *script* est la référence du fichier script.

 Rappelons que comme pour toute commande externe, la deuxième forme requiert que si le chemin du script ne figure pas dans la variable **PATH**, il faut que *script* contienne un / (comme dans `./monscript`).

Chaque *argument* est un mot. Si ce mot contient des blancs, il faut les protéger. À l'intérieur du script, on peut faire référence aux arguments en utilisant les paramètres positionnels (comme dans les fonctions) :

- **\$1** est remplacé par le premier argument
- **\$2** est remplacé par le deuxième argument
- etc. jusqu'à **\$9**
- à partir du dixième argument, il faut utiliser **\${n}** (ex : **\${10}**)

Outre ces paramètres positionnels, il existe aussi les **paramètres spéciaux** **\$\*** et **\$@**, qui sont remplacés par la liste de tous les arguments. La différence entre **\*** et **@** est la même que pour les tableaux (voir section [15.B.3.a](#) page 219) : elle n'existe qu'entre guillemets.

**Aucun des paramètres positionnels n'est modifiable**<sup>1</sup>. Ils sont les mêmes que pour les fonctions<sup>2</sup>.

Il existe aussi les paramètres spéciaux suivants :

- **##** : remplacé par le nombre d'arguments passés au script ou à la fonction ;

1. Il s'agit là d'un tout petit mensonge...

2. On n'avait pas encore introduit **\$@** avec les fonctions au chapitre 9.

- **\$\$** : remplacé par le numéro du processus (son PID) courant, notamment celui qui exécute le script ou la fonction. Ce paramètre sert souvent pour créer des fichiers temporaires utiles au script dans le répertoire /tmp. Par exemple, on pourra créer le fichier /tmp/fic\$\$\$. Si plusieurs processus exécutent ce script, chacun créera un fichier de nom différent ;
- **\$0** : n'existe que dans les scripts<sup>3</sup> et est remplacé par *script* (la référence par laquelle le script a été désigné lors de son exécution).

## Exemples

Supposons que *monscript* contienne le texte suivant :

```
#!/bin/bash

echo "Script appelé par : $0 (contenu dans \$0)"
echo "avec $# argument(s)"

echo "Ses arguments sont :"
echo "\$1 : $1"
echo "\$2 : $2"
echo "\$3 : $3"
echo "\$4 : $4"

echo "Utilisation de \$@ et \$* sans guillemets"
echo "\$@ : " $@
echo "\$* : " $*

echo "Utilisation de \$@ et \$* avec guillemets (IFS=,)"
IFS=', '
echo "\$@ : " "$@"
echo "\$* : " "$*"
```

voici deux exemples d'exécution :

```
$ ./monscript arg1 arg2 arg3
Script appelé par : ./monscript (contenu dans $0)
avec 3 argument(s)
Ses arguments sont :
$1 : arg1
$2 : arg2
$3 : arg3
$4 :
Utilisation de $@ et $* sans guillemets
$@ : arg1 arg2 arg3
$* : arg1 arg2 arg3
Utilisation de $@ et $* avec guillemets (IFS=,)
$@ : arg1 arg2 arg3
$* : arg1,arg2,arg3
```

⇒ on voit que **\$4** est vide (non initialisée). On rappelle que pour les deux dernière lignes "**\$@**" est remplacé par les 3 chaînes "*arg1*" "*arg2*" "*arg3*", alors que "**\$\***" est remplacé par la chaîne "*arg1,arg2,arg3*" (la virgule venant de **IFS**)

```
$ ./monscript arg1 "arg2____sur____plusieurs mots" arg3
Script appelé par : ./monscript (contenu dans $0)
avec 3 argument(s)
Ses arguments sont :
$1 : arg1
$2 : arg2____sur____plusieurs mots
```

3. En réalité, il existe aussi dans un shell bash, mais dans ce cas il contient généralement `-bash`.

```

$3 : arg3
$4 :
Utilisation de $@ et $* sans guillemets
$@ : arg1 arg2 sur plusieurs mots arg3
$* : arg1 arg2 sur plusieurs mots arg3
Utilisation de $@ et $* avec guillemets (IFS=,)
$@ : arg1 arg2 sur plusieurs mots arg3
$* : arg1, arg2 sur plusieurs mots, arg3

```

□

## 16.A.5 Substitutions de sous-chaînes et paramètres \$@ et \$\*

Toutes les substitutions de sous-chaînes peuvent s'appliquer à l'ensemble des paramètres positionnels, en utilisant les paramètres spéciaux **\$@** et **\$\***. Plutôt que *identificateur* dans leur définition il faut écrire **@** ou **\***. La différence entre les deux écritures reposant une fois encore sur l'utilisation des guillemets.

Les substitutions de sous-chaînes avec les motifs (voir section [15.A.4.c page 216](#)) pour l'ensemble des paramètres positionnels s'appliquent à chaque paramètre positionnel et ne méritent pas d'explications supplémentaires.


En revanche, la substitution d'une sous-chaîne (voir section [15.A.4.b page 215](#)) appliquée à **\$@** et **\$\*** est, comme pour les tableaux, un cas particulier. En effet, le résultat des substitutions suivantes :

**\${@ : décalage [ : longueur ] }**

et

**\${\* : décalage [ : longueur ] }**

est d'obtenir la liste des *longueur* paramètres positionnels à partir du *décalage*<sup>ième</sup> paramètre positionnel. *longueur* peut être omis, auquel cas la liste va jusqu'au dernier paramètre positionnel. **@** et **\*** ont toujours la signification qu'on leur connaît. *décalage* peut être négatif pour partir de la fin des paramètres positionnels (le dernier élément portant le numéro -1).

 Une différence avec les tableaux est que le premier paramètre positionnel porte le numéro 1 (les tableaux commençant à l'indice 0). De plus, il n'y a pas de "trous" dans les paramètres positionnels.

### Exemple

Supposons que *monscript* contienne le texte suivant :

```

#!/bin/bash

echo "il y a $# paramètres positionnels"

IFS=','
echo 'résultat de ${@:2:4} : ' "${@:2:4}"
echo 'résultat de ${*:2:4} : ' "${*:2:4}"
echo 'résultat de ${@: -3:2} : ' "${@: -3:2}"
echo 'résultat de ${*: -3:2} : ' "${*: -3:2}"

```

voici un exemple d'exécution :

```
$ ./monscript a b c d e f g h
il y a 8 paramètres positionnels
résultat de ${@:2:4}      : b c d e
résultat de ${*:2:4}      : b,c,d,e
résultat de ${@: -3:2}    : f g
résultat de ${*: -3:2}    : f,g
```



## 16.A.6 Paramètres et variables dans les scripts et fonctions

### 16.A.6.a Paramètres dans les scripts et fonctions

Les paramètres (positionnels et spéciaux) peuvent être utilisés dans les scripts et les fonctions. Lorsqu'une fonction est créée (utilisée) dans un script, celle-ci dispose de ses propres paramètres positionnels, ainsi que de ses propres paramètres spéciaux `$#`, `$@` et `$*`. En revanche, les paramètres spéciaux `$0` et `$$` restent ceux du script.

#### Exemple

Supposons que *monscript* contienne le texte suivant :

```
#!/bin/bash

function mafonction {

    echo -e "\n\tDans la fonction mafonction :"
    echo -e "\t\t$0 vaut : $0"
    echo -e "\t\t$$ vaut : $$"
    echo -e "\t\t$# vaut : $#"
```

```
    echo -e "\t\t$1 vaut : $1"
    echo -e "\t\t$2 vaut : $2"
    echo -e "\t\t$3 vaut : $3"
    echo -e "\t\t$* vaut : $*"

} # mafonction()

echo -e "\nDans le script : "
```

```
echo "\$0 vaut : $0"
echo "\$$ vaut : $$"
echo "\$# vaut : $#"
```

```
echo "\$1 vaut : $1"
echo "\$2 vaut : $2"
echo "\$3 vaut : $3"
echo "\$* vaut : $*"

mafonction autorearg1 autorearg2

echo -e "\nAprès l'appel de mafonction :"
```

```
echo "\$0 vaut : $0"
echo "\$$ vaut : $$"
echo "\$# vaut : $#"
```

```
echo "\$1 vaut : $1"
echo "\$2 vaut : $2"
echo "\$3 vaut : $3"
echo "\$* vaut : $*"

```

voici un exemple d'exécution :

```
$ ./monscript arg1 arg2 arg3
Dans le script :
$0 vaut : ./monscript
$$ vaut : 25084
$# vaut : 3
$1 vaut : arg1
```

```
$2 vaut : arg2
$3 vaut : arg3
$* vaut : arg1 arg2 arg3
```

```
Dans la fonction mafonction :
$0 vaut : ./monscript
$$ vaut : 25084
$# vaut : 2
$1 vaut : autorearg1
$2 vaut : autorearg2
$3 vaut :
$* vaut : autorearg1 autorearg2
```

Après l'appel de mafonction :

```
$0 vaut : ./monscript
$$ vaut : 25084
$# vaut : 3
$1 vaut : arg1
$2 vaut : arg2
$3 vaut : arg3
$* vaut : arg1 arg2 arg3
```



### 16.A.6.b Portée des variables et tableaux dans les scripts et fonctions

Les variables et les tableaux créés dans un script **sont globaux**. S'ils ont été créés avant l'appel d'une fonction, alors ils sont connus dans la fonction lorsqu'elle est appelée. De même, les variables et les tableaux créés dans une fonction sont connus dans le script après l'appel de cette fonction.

#### Exemple

Supposons que *monscript* contienne le texte suivant :

```
#!/bin/bash

function fonc_1 {
    varfonc_1='contenu de varfonc_1'
    echo -e '\n\tDans la fonction fonc_1'
    echo -e "\tvarscript : ${varscript-variable non initialisée}"
    echo -e "\tvarfonc_1 : ${varfonc_1-variable non initialisée}"
    echo -e "\tvarfonc_2 : ${varfonc_2-variable non initialisée}"
}

function fonc_2 {
    varfonc_2='contenu de varfonc_2'
    echo -e '\n\tDans la fonction fonc_2'
    echo -e "\tvarscript : ${varscript-variable non initialisée}"
    echo -e "\tvarfonc_1 : ${varfonc_1-variable non initialisée}"
    echo -e "\tvarfonc_2 : ${varfonc_2-variable non initialisée}"
}

varscript='contenu de varscript'
echo -e '\nAvant les appels des fonctions :'
echo "varscript : ${varscript-variable non initialisée}"
echo "varfonc_1 : ${varfonc_1-variable non initialisée}"
echo "varfonc_2 : ${varfonc_2-variable non initialisée}"

fonc_1
fonc_2

echo -e '\nAprès les appels des fonctions :'
echo "varscript : ${varscript-variable non initialisée}"
echo "varfonc_1 : ${varfonc_1-variable non initialisée}"
echo "varfonc_2 : ${varfonc_2-variable non initialisée}"
```



voici un exemple d'exécution :

```
$ ./monscript
```

```
Avant les appels des fonctions :
varscript : contenu de varscript
varfonc_1 : variable non initialisée
varfonc_2 : variable non initialisée
```

```
Dans la fonction fonc_1
varscript : contenu de varscript
varfonc_1 : contenu de varfonc_1
varfonc_2 : variable non initialisée
```

```
Dans la fonction fonc_2
varscript : contenu de varscript
varfonc_1 : contenu de varfonc_1
varfonc_2 : contenu de varfonc_2
```

```
Après les appels des fonctions :
varscript : contenu de varscript
varfonc_1 : contenu de varfonc_1
varfonc_2 : contenu de varfonc_2
```

⇒ on remarque notamment qu'une variable créée dans une fonction qui a été appelée devient ensuite connue, y compris dans les autres fonctions.

□

✍ Si une fonction modifie une variable globale, alors lorsque la fonction se termine, la modification reste effective.

### 16.A.6.c local : créer des variables et des tableaux locaux

Une alternative est d'utiliser la commande interne (instruction) **local**.

#### Synopsis

```
local identificateur [=mot-ou-tableau] { identificateur [=mot-ou-tableau] }
```

où *mot-ou-tableau* doit prendre l'une des formes suivantes :

<i>mot</i>	<i>(pour créer une variable locale)</i>
<b>( { mot } )</b>	<i>(pour créer un tableau local)</i>

L'instruction **local** n'est utilisable qu'à l'intérieur d'une fonction et permet de créer des variables et des tableaux locaux *identificateur* qui **masquent** les variables et les tableaux globaux de même nom. Toutes les modifications apportées à ces variables ne sont effectives que dans la fonction.

#### Exemple

Supposons que *monscript* contienne le texte suivant :

```
#!/bin/bash

function mafonction {

    var1='valeur de var1 dans mafonction'
    tab1=(éléments de tab1 dans mafonction)
    local var2='valeur de var2 dans mafonction'
    local tab2=(éléments de tab2 dans mafonction)

    echo -e "\n\tDans la fonction mafonction : "
    echo -e '\t$var1 contient : ' $var1
    echo -e '\t$tab1 contient : ' ${tab1[*]}
    echo -e '\t$var2 contient : ' $var2
    echo -e '\t$tab2 contient : ' ${tab2[*]}

} # mafonction()

var1="valeur de var1 dans script"
tab1=(éléments de tab1 dans script)
var2="valeur de var2 dans script"
tab2=(éléments de tab2 dans script)

echo -e "\nDans le script : "
echo '$var1 contient : ' $var1
echo '$tab1 contient : ' ${tab1[*]}
echo '$var2 contient : ' $var2
echo '$tab2 contient : ' ${tab2[*]}

mafonction

echo -e "\nAprès l'appel de mafonction : "
echo '$var1 contient : ' $var1
echo '$tab1 contient : ' ${tab1[*]}
echo '$var2 contient : ' $var2
echo '$tab2 contient : ' ${tab2[*]}
```

*Son exécution montre que les variables **var2** et **tab2** ne sont pas affectées par les modifications des variables locales de même nom dans la fonction **mafonction** :*

\$ ./monsript

Dans le script :

```
$var1 contient : valeur de var1 dans script
$tab1 contient : éléments de tab1 dans script
$var2 contient : valeur de var2 dans script
$tab2 contient : éléments de tab2 dans script
```

Dans la fonction mafonction :

```
$var1 contient : valeur de var1 dans mafonction
$tab1 contient : éléments de tab1 dans mafonction
$var2 contient : valeur de var2 dans mafonction
$tab2 contient : éléments de tab2 dans mafonction
```

Après l'appel de mafonction :


```
$var1 contient : valeur de var1 dans mafonction
$tab1 contient : éléments de tab1 dans mafonction
$var2 contient : valeur de var2 dans script
$tab2 contient : éléments de tab2 dans script
```




## 16.B Valeur de retour et suites de commandes

### 16.B.1 Valeur de retour des commandes

Toute commande, de quelque nature que ce soit, qui se termine sur un système Unix renvoie une valeur de retour (ou code de retour) qui n'est pas affichée mais que le shell affecte au paramètre spécial `$?`. Cette valeur est codée sur un octet et est comprise entre 0 et 255.


 Par convention, la valeur de retour 0 devrait indiquer que la commande s'est déroulée normalement. Une valeur de retour différente de 0 devrait indiquer une erreur.

 Comme toute convention, celle-ci doit être suivie le plus souvent possible. On peut cependant imaginer des cas où il peut être pratique de ne pas la suivre.

Plus spécifiquement, certaines commandes retourneront une valeur de retour 1 en cas d'échec, et une valeur de retour 2 si elle est mal employée, par exemple une erreur dans ses arguments ou options, mais ce n'est pas une règle.


Par exemple, la commande `cd` retourne 0 si elle réussit et 1 dans les autres cas. La commande `ls` retourne 0 si elle réussit. Selon le système (par exemple, la Mandriva 2007), elle retourne 1 si elle ne peut fournir les renseignements demandés pour au moins un fichier ou répertoire en arguments, et 2 si une mauvaise option est utilisée. Sur d'autres systèmes (comme Debian), elle retournera 2 quelle que soit l'erreur.

La documentation de toute commande devrait indiquer la signification de ses valeurs de retour. C'est en principe le cas du manuel en ligne.

 La valeur de retour d'une commande est principalement exploitée dans les scripts shell, notamment lorsque l'exécution d'une commande est conditionnée par le succès d'une (suite de) commande(s) précédente.

### Exemples (exécutés sur une Debian)

```
$ rm fichier
```

 suppression réussie d'un fichier

```
$ echo $?
```

```
0
```

 la valeur de retour est bien 0


```
$ rm /bin/ls
```

```
rm: détruire un fichier protégé en écriture fichier régulier '/bin/ls'? y
rm: ne peut enlever '/bin/ls': Permission non accordée
```

 échec de la suppression d'un fichier

```
$ echo $?
```

```
1
```

 la valeur de retour est bien différente de 0

```
$ echo $?
0
```

⇒ cette fois la valeur de retour est 0 car la commande précédente (**echo**) a réussi

```
$ cd /tmp
$ echo $?
0
```

⇒ changement de répertoire réussi

```
$ cd /root
-bash: cd: /root: Permission non accordée
$ echo $?
1
```

⇒ changement de répertoire impossible

```
$ ls . /root
.:
args fic1 fic2 fic3 fic4 fic5 monscript portee rep1 rep2
ls: /root: Permission non accordée
$ echo $?
2
```

⇒ Impossibilité de lire le contenu d'un répertoire en arguments qui conduit à une valeur de retour à 2, alors que le contenu du répertoire courant est affiché avec succès

```
$ ls -z ~
ls: option invalide -- z
Pour en savoir davantage, faites: « ls --help ».
$ echo $?
2
```

□

### 16.B.1.a return : indiquer une valeur de retour pour une fonction

Une fonction retourne aussi une valeur de retour : celle de la dernière (suite de) commande(s) exécutée. L'instruction **return** permet de spécifier une valeur de retour.

#### Synopsis

**return** [*n*]

Termine la fonction en cours et fixe à *n* sa valeur de retour. Si *n* n'est pas précisé, la valeur de retour est celle de la dernière (suite de) commande(s) exécutée.

✍ **return** n'est normalement utilisable que dans une fonction. Elle peut être néanmoins utilisée hors fonction dans un script, à condition que celui-ci soit exécuté par la commande **source**, sinon cela produit une erreur.


### 16.B.1.b **exit** : indiquer une valeur de retour pour un script

Un script retourne aussi une valeur de retour : celle de la dernière (suite de) commande(s) exécutée. L'instruction **exit** permet de spécifier une valeur de retour.

#### **Synopsis**

**exit** [*n*]

**exit** peut être utilisée n'importe où dans un script, y compris dans une fonction. Elle termine le script en cours et fixe à *n* sa valeur de retour. Si *n* n'est pas précisé, la valeur de retour est celle de la dernière (composition de) commande(s) exécutée.

 D'une façon plus générale, **exit** permet de terminer le shell courant, même le shell interactif depuis lequel on tape des commandes, où taper `Ctrl-d` en début de ligne produit le même effet.


### 16.B.1.c Valeur de retour d'un pipeline

On rappelle qu'un *pipeline* (voir section 6.A page 73) est un ensemble de commandes séparées par un `|`, ayant la forme suivante :

*commande*<sub>1</sub> | *commande*<sub>2</sub> | ... | *commande*<sub>*n*</sub>

où les commandes sont exécutées en parallèle et l'entrée d'une commande est connectée à la sortie de la commande située à sa gauche.

Du point de vue de `bash`, un pipeline est vu comme une seule commande dont la valeur de retour est celle de la dernière commande du pipeline.

 L'activation de l'option *pipefail* (par `shopt -os pipefail`) permet de modifier ce comportement. Si elle est activée, la valeur de retour du pipeline est celle de la commande la plus à droite dont la valeur de retour n'est pas 0, et 0 si toutes les commandes se terminent avec succès.

## 16.B.2 Négation de la valeur de retour d'une commande ou d'un pipeline

Il peut parfois être utile que la valeur de retour retenue d'une commande ou d'un pipeline soit l'inverse de sa véritable valeur de retour. Pour cela, il faut placer le caractère spécial **!** en début de commande ou du pipeline.

#### **Synopsis**

**!** *commande-ou-pipeline*



**Il faut impérativement au moins un blanc entre le ! et la commande ou le pipeline.**


La valeur de retour de **!** *commande-ou-pipeline* est :

- 1 si *commande-ou-pipeline* renvoie 0 ;
- 0 dans les autres cas.

### 16.B.3 Suites de commandes

Sur une ligne de commandes, il est possible d'exécuter plusieurs commandes ou pipelines, en écrivant une suite de commandes. On verra plus tard, notamment en présentant les contrôles de flux, une utilisation de ces suites.

Une suite de commande est une séquence de plusieurs commandes ou pipelines séparés par l'un des opérateurs suivants : `;`, `&`, `&&` et `||`. Ces opérateurs sont donc des séparateurs de commandes et de pipelines.


 Dans la suite du document, *commande* fera référence à une commande simple ou un pipeline, éventuellement précédé de la négation (!).

On a déjà vu les opérateurs `;` et `&` précédemment :

- `;` est utilisé pour l'exécution séquentielle de deux commandes. La suite :

*commande<sub>1</sub> ; commande<sub>2</sub>*


veut dire que *commande<sub>2</sub>* sera exécutée lorsque *commande<sub>1</sub>* sera terminée ;

 Ainsi, `;` et le retour à la ligne jouent le même rôle : ils sont interchangeables.

- `&` est utilisé pour l'exécution en tâche de fond d'une commande. La suite :

*commande<sub>1</sub> & commande<sub>2</sub>*

veut dire que *commande<sub>1</sub>* est exécutée en tâche de fond (en parallèle) et que *commande<sub>2</sub>* est exécutée sans attendre la fin de *commande<sub>1</sub>*.

 Puisque bash n'attend pas la terminaison de *commande<sub>1</sub>*, la valeur de retour de *commande<sub>1</sub> &* est toujours 0.

En revanche, les opérateurs `&&` et `||` sont nouveaux et servent pour des exécutions conditionnelles :

- `&&` est utilisé pour conditionner l'exécution d'une commande **au succès** de la précédente. La suite :

*commande<sub>1</sub> && commande<sub>2</sub>*

veut dire que *commande<sub>2</sub>* ne sera exécutée que si *commande<sub>1</sub>* se termine avec une valeur de retour à 0 ;

- `||` est utilisé pour conditionner l'exécution d'une commande **à l'échec** de la précédente. La suite :

*commande<sub>1</sub> || commande<sub>2</sub>*

veut dire que *commande<sub>2</sub>* ne sera exécutée que si *commande<sub>1</sub>* se termine avec une valeur de retour différente de 0.

Une suite de commandes a donc la forme générale suivante :

*commande { opérateur commande }*

et peut être terminée par un `;`, un `&` ou un retour à la ligne.

Puisque plusieurs *opérateurs* peuvent figurer dans une suite, il faut leur donner une priorité : `&&` et `||` ont la même priorité qui est supérieure à la priorité de `;` et `&`, qui est la même. L'associativité, qui n'a d'importance ici que pour `&&` et `||`, est gauche-droite.

De ce fait, si l'on écrit quelque chose d'aussi illisible que :


$$cmd_1 ; cmd_2 \& cmd_3 \&\& cmd_4 || cmd_5 \&\& cmd_6 ; cmd_7$$

alors, les opérateurs sont associés comme l'indique l'imbrication des boîtes ci-dessous :

$$cmd_1 ; \left[ cmd_2 \& \left[ \left[ cmd_3 \&\& cmd_4 \right] || cmd_5 \&\& cmd_6 \right] ; cmd_7 \right]$$


et  $cmd_6$  ne sera exécutée que si :

- $cmd_3$  et  $cmd_4$  retournent vrai
- ou  $cmd_5$  retourne vrai


 La valeur de retour d'une suite de commandes est celle de la dernière commande exécutée. Dans la suite précédente, ce sera celle de  $cmd_7$ .

### Exemples

```
$ cd /root && echo "placé dans /root"
-bash: cd: /root: Permission non accordée
$ echo $?
1
```

 la dernière commande exécutée est **cd** qui a échoué

```
$ cd /root || echo "impossible d'aller dans /root"
-bash: cd: /root: Permission non accordée
impossible d'aller dans /root
$ echo $?
0
```

 la dernière commande exécutée est **echo** qui a réussi

*Même chose en se débarrassant de la sortie d'erreur de cd*


```
$ cd /root 2> /dev/null && echo "placé dans /root"
$ echo $?
1
$ cd /root 2> /dev/null || echo "impossible d'aller dans /root"
impossible d'aller dans /root
$ echo $?
0
```

### Vérification des priorités

```
$ true || true && false
$ echo $?
1
```

 cela renvoie faux car la priorité est gauche-droite et on a évalué **(true || true) && false**

```
$ true || (true && false)
$ echo $?
0
```

 alors qu'on aurait eu ce résultat si la priorité était droite-gauche.

□

## 16.C Tester des conditions


Il existe des commandes et des constructions qui n'ont qu'un objet : retourner la valeur 0 ou 1, selon que la condition testée est vraie ou fausse.

### 16.C.1 Tester des conditions arithmétiques

Bash fournit une construction spéciale destinée à tester des conditions arithmétiques. Cette construction est la suivante :

`( (expression) )`

où *expression* est une expression arithmétique, comme dans l'instruction **let**<sup>4</sup> (voir section 15.A page 207).

 Une différence importante et fort appréciable avec l'expression arithmétique de **let**, est que dans `( (expression) )`, il n'est pas nécessaire de protéger les caractères spéciaux comme **&**, **<**, **>**, etc.

Cette construction est considérée comme une commande (une instruction). Elle est utilisable partout où une commande est admise, c'est à dire dans une suite de commandes. **Elle n'affiche rien** (sauf éventuellement sur la sortie d'erreur si la syntaxe de l'*expression* est incorrecte). Simplement, elle retourne 0 si l'*expression* est vraie, et 1 sinon (même en cas d'erreur).


Comme en C/C++ ou Java et autres langages, une comparaison ou une expression booléenne à l'intérieur de *expression* vaut 0 si elle est fausse et 1 si elle est vraie. De même, l'*expression* sera vraie si sa valeur est différente de 0, et fausse si elle vaut 0.




**On remarque que le vrai booléen (1) et le faux booléen (0) ont exactement le sens inverse de la valeur retournée par cette instruction. Mais c'est elle qui se charge d'effectuer la conversion. Donc utilisez `expression` comme vous le feriez en C/C++.**

### Exemples


```
$ ( (3 < 2) )
$ echo $?
1
```

 l'expression `3 < 2` est bien fausse. On remarque qu'il n'est pas nécessaire de protéger le **<** (mais ce n'est pas interdit). De plus, les blancs ne sont pas obligatoires et ne servent qu'à la lisibilité

```
$ ( (3 > 2) )
$ echo $?
0
```

 l'expression `3 > 2` est bien vraie

```
$ ( (x = 3 < 2) )
$ echo $? $x
1 0
```

 on voit bien la conversion : alors que **x** vaut bien 0 (résultat de `3 < 2` à l'intérieur de l'expression), la valeur de retour est 1

4. En réalité, cette construction est équivalente à l'instruction **let** *expression*.



```
$ ((x = 3 > 2))
$ echo $? $x
0 1
```

⇒ autre conversion : alors que **x** vaut bien 1 (résultat de  $3 > 2$  à l'intérieur de l'expression), la valeur de retour est 0

```
$ x=5
$ ((x>0)) && echo "x est positif"
x est positif
$ ((0 < x && x < 10)) && echo "x est compris dans l'intervalle ]0,10["
x est compris dans l'intervalle ]0,10[
$ ((y = 0 < x && x < 10)) && echo "x est compris dans l'intervalle ]0,10["
x est compris dans l'intervalle ]0,10[
$ echo $y
1
```

□

## 16.C.2 test : tester des conditions de différentes natures

À l'instar de l'instruction précédente, la commande interne<sup>5</sup> **test** est une commande muette (si elle est syntaxiquement correcte) qui renvoie 0 si la condition testée est vraie, et 1 sinon. Il existe deux syntaxes équivalentes pour cette commande.

### Synopsis

```
test condition
ou
[ condition ]
```



**Cette commande a une syntaxe très rigide. Déjà, il doit obligatoirement y avoir au moins un blanc qui sépare la condition des crochets.**

Les conditions testées peuvent être de natures très différentes : tests sur des fichiers, des chaînes, des entiers, ou une combinaison de tout ça. Pour chaque catégorie, la syntaxe de *condition* est indiquée à gauche et sa signification à droite. La liste est presque exhaustive, se reporter au manuel de bash (partie CONDITIONAL EXPRESSIONS dans le manuel en anglais de bash) pour les avoir toutes.



**Tous les termes des conditions ou des combinaisons suivantes doivent être impérativement séparés par des blancs.**

5. Comme de nombreuses commandes internes de bash, il existe aussi la commande externe **test** qui a la même fonction, mais reconnaît moins de conditions. Ces commandes internes existent à des fins d'efficacité.

## Conditions sur les fichiers

Dans ce qui suit, « fichier » est à prendre au sens large : un fichier de tout type.

### Tests sur l'existence et le type d'un fichier :

- e** *référence* vrai ssi le fichier *référence* existe
- f** *référence* vrai ssi le fichier *référence* existe et est un fichier ordinaire
- d** *référence* vrai ssi le fichier *référence* existe et est un répertoire
- c** *référence* vrai ssi le fichier *référence* existe et est un fichier spécial en mode caractère
- b** *référence* vrai ssi le fichier *référence* existe et est un fichier spécial en mode bloc
- h** *référence* vrai ssi le fichier *référence* existe et est un fichier spécial lien symbolique
- p** *référence* vrai ssi le fichier *référence* existe et est un fichier spécial tube nommé
- S** *référence* vrai ssi le fichier *référence* existe et est un fichier spécial socket

### Tests sur les permissions d'un fichier :

- r** *référence* vrai ssi le fichier *référence* existe et l'utilisateur exécutant la commande a le droit de lecture sur *référence*
- w** *référence* vrai ssi le fichier *référence* existe et l'utilisateur exécutant la commande a le droit d'écriture sur *référence*
- x** *référence* vrai ssi le fichier *référence* existe et l'utilisateur exécutant la commande a le droit d'exécution sur *référence*
- u** *référence* vrai ssi le fichier *référence* existe et a son *bit set-uid* positionné<sup>6</sup>
- g** *référence* vrai ssi le fichier *référence* existe et a son *bit set-gid* positionné<sup>6</sup>
- k** *référence* vrai ssi le fichier *référence* existe et a son *sticky-bit* positionné<sup>6</sup>

### Tests sur l'appartenance d'un fichier :

- O** *référence* vrai ssi le fichier *référence* existe et est propriété de l'utilisateur exécutant la commande
- G** *référence* vrai ssi le fichier *référence* existe et appartient à l'un des groupes de l'utilisateur exécutant la commande

### Autres tests sur un fichier :

- s** *référence* vrai ssi le fichier *référence* existe et sa taille est supérieure à 0
- N** *référence* vrai ssi le fichier *référence* existe et a été modifié depuis sa dernière lecture

### Comparaison de deux fichiers :

- référence*<sub>1</sub> **-nt** *référence*<sub>2</sub> vrai ssi la date de modification de *référence*<sub>1</sub> est plus récente que celle de *référence*<sub>2</sub>, ou *référence*<sub>1</sub> existe mais pas *référence*<sub>2</sub>
- référence*<sub>1</sub> **-ot** *référence*<sub>2</sub> vrai ssi la date de modification de *référence*<sub>1</sub> est plus ancienne que celle de *référence*<sub>2</sub>, ou *référence*<sub>2</sub> existe mais pas *référence*<sub>1</sub>
- référence*<sub>1</sub> **-ef** *référence*<sub>2</sub> vrai ssi *référence*<sub>1</sub> et *référence*<sub>2</sub> désignent le même périphérique ou ont le même numéro d'*inode* (voir section 14.A.1 page 191)

6. Les bits *set-uid*, *set-gid* et *sticky* seront vus section 14.B page 197.

## Conditions sur les chaînes

$chaîne_1 = chaîne_2$	vrai ssi les deux chaînes sont identiques
$chaîne_1 == chaîne_2$	vrai ssi les deux chaînes sont identiques
$chaîne_1 != chaîne_2$	vrai ssi les deux chaînes sont différentes
$chaîne_1 < chaîne_2$	vrai ssi $chaîne_1$ est inférieure lexicographiquement à $chaîne_2$ . <b>Le caractère &lt; étant spécial, il faut le protéger.</b>
$chaîne_1 > chaîne_2$	vrai ssi $chaîne_1$ est supérieure lexicographiquement à $chaîne_2$ . <b>Le caractère &gt; étant spécial, il faut le protéger.</b>
<code>-n chaîne</code>	vrai ssi $chaîne$ n'est pas vide (contient au moins un caractère)
<code>-z chaîne</code>	vrai ssi $chaîne$ est vide (ne contient aucun caractère)



Pour ces deux derniers tests, une erreur fréquente consiste à tester le contenu d'une variable sans l'encadrer par des guillemets. En effet, si la variable `var` est inexistante ou vide, alors le test :

```
test -n $var
```

retourne vrai !

Sans rentrer dans les détails, après substitution de `$var`, le test devient :

```
test -n
```

or, si `test` n'a qu'un argument (ici `-n`) , alors il retourne vrai si cet argument n'est pas vide !

La bonne utilisation de ce test est d'encadrer la substitution de la variable par des guillemets. Dans ce cas, le test :

```
test -n "$var"
```


deviendra :

```
test -n ""
```

et retourne bien faux.

## Conditions sur les entiers

$entier_1 -lt entier_2$	(less than) vrai ssi $entier_1$ est strictement inférieur à $entier_2$
$entier_1 -le entier_2$	(less or equal) vrai ssi $entier_1$ est inférieur ou égal à $entier_2$
$entier_1 -eq entier_2$	(equal) vrai ssi $entier_1$ et $entier_2$ sont égaux
$entier_1 -ne entier_2$	(not equal) vrai ssi $entier_1$ et $entier_2$ sont différents
$entier_1 -ge entier_2$	(greater or equal) vrai ssi $entier_1$ est supérieur ou égal à $entier_2$
$entier_1 -gt entier_2$	(greater than) vrai ssi $entier_1$ est strictement supérieur à $entier_2$

 On notera que la construction `(( expression ))` est beaucoup plus adaptée et naturelle pour les tests sur les entiers...


## Connecteurs logiques des conditions

Afin de combiner les conditions, les connecteurs logiques et les parenthèses peuvent être utilisés :


- `! condition` (négation) vrai ssi *condition* est fausse
- `condition1 -o condition2` (ou) vrai ssi l'une ou l'autre des conditions est vraie
- `condition1 -a condition2` (et) vrai ssi les deux conditions sont vraies
- `( condition )` permet de grouper des conditions. **La condition doit être séparée des parenthèses par au moins un blanc. De plus, les parenthèses sont des caractères spéciaux pour bash : il faut les protéger.**

## Exemples


```
$ test -f /bin/ls
$ echo $?
0
```

 */bin/ls est un fichier ordinaire existant*


```
$ test -f /bin/ls -a -x /bin/ls ; echo $?
0
```

 */bin/ls est un fichier ordinaire existant et exécutable par l'utilisateur*


```
$ test -f /bin/ls -a -w /bin/ls ; echo $?
1
```

 */bin/ls est un fichier ordinaire existant mais pas modifiable (du moins pour l'utilisateur ayant tapé la commande)*


```
$ [ -d /bin/ls ] ; echo $?
1
```

 */bin/ls n'existe pas ou, s'il existe, n'est pas un répertoire*


```
$ var=5
$ [ ( $var -eq 3 ) -o ( $var -gt 10 ) ]
-bash: syntax error near unexpected token `$(var'
```

 *erreur : il faut protéger les parenthèses !*

```
$ [ \ ( $var -eq 3 \ ) -o \ ( $var -gt 10 \ ) ]
$ echo $?
1
```

 *var ne contient pas 3 et n'est pas supérieur strictement à 10. Il aurait quand même été plus adapté d'utiliser la forme `((var == 3 || var > 10))`.*

```
$ [ 123 == 123 ] ; echo $?
0
```

 *les chaînes 123 et 123 sont identiques.*

```
$ [ 123 -eq 0123 ] ; echo $?
0
```

⇒ les **entiers** *123* et *0123* sont égaux

```
$ [ 123 == 0123 ] ; echo $?
1
```

⇒ mais pas les **chaînes** *123* et *0123* !

```
$ [ 123 < 0123 ]
-bash: 0123: Aucun fichier ou répertoire de ce type
```

⇒ erreur : il faut protéger < !

```
$ [ 123 \< 0123 ] ; echo $?
1
```

⇒ la chaîne *123* n'est pas inférieure lexicographiquement à la chaîne *0123*

```
$ [ 123 \> 0123 ] ; echo $?
0
```

⇒ la chaîne *123* est supérieure lexicographiquement à la chaîne *0123*. Ceci parce que le caractère *1* est supérieur au caractère *0*.

□

### 16.C.3 Les commandes true et false

Ce sont des commandes internes (qui existent aussi comme commandes externes) qui ne font que retourner une valeur de retour :

- **true** retourne toujours la valeur de retour 0
- **false** retourne toujours la valeur de retour 1

## 16.D Contrôle de flux pour la programmation bash

On aborde maintenant les structures de contrôle de flux du langage de programmation bash : instructions conditionnelles, répétitives, branchements conditionnels et menus. Dans ce qui suit, le terme *liste* désigne une suite de commandes (voir section [16.B.3 page 240](#)).

### 16.D.1 L'instruction if

L'instruction conditionnelle en bash est l'instruction **if**. Elle exécute des instructions selon la valeur de retour d'une suite de commandes.

#### Synopsis

```
if liste ; then liste ; { elif liste ; then liste ; } [ else liste ; ] fi
```

où tous les ; peuvent être remplacés par (ou suivis d') un nombre quelconque de retours à la ligne. Aussi, les mots clés **if**, **then**, **elif** et **else** peuvent être suivis de retours à la ligne.

✍ C'est aussi le cas pour toutes les structures de contrôle qui vont suivre (**for**, **while**, **until**, **case** et **select**) et on ne les présentera généralement qu'écrits sur une seule ligne.

On peut donc réécrire cette instruction ainsi :

<pre> <b>if</b> liste <b>then</b>     liste {     <b>elif</b> liste     <b>then</b>         liste } [     <b>else</b>         liste ] <b>fi</b> </pre>	ou encore	<pre> <b>if</b> liste ; <b>then</b>     liste {     <b>elif</b> liste ; <b>then</b>         liste } [     <b>else</b>         liste ] <b>fi</b> </pre>
--	-----------	--

Comme on le voit, il peut y avoir un nombre quelconque de constructions **elif** *liste* ; **then** *liste* ; . C'est une contraction de **else if...then...** qui ne nécessite pas de **fi**. La dernière partie **else...** est aussi optionnelle.

Prenons le cas le plus simple où il n'y a pas de **elif** ni de **else** :

**if** *liste* ; **then** *liste* ; **fi**

D'abord la *liste* qui suit **if** est exécutée. Si sa valeur de retour est 0, alors la *liste* qui suit **then** est aussi exécutée, sinon bash passe au **fi** qui termine le **if**.

S'il y un **else** comme dans :

**if** *liste* ; **then** *liste* ; **else** *liste* ; **fi**


alors la *liste* qui suit **else** n'est exécutée que si la valeur de retour de la *liste* qui suit **if** est différente de 0.

Si à la place de **else** il y a **elif** comme dans :

**if** *liste*<sub>1</sub> ; **then** *liste*<sub>2</sub> ; **elif** *liste*<sub>3</sub> ; **then** *liste*<sub>4</sub> ; **fi**

alors si *liste*<sub>1</sub> a une valeur de retour différente de 0, alors *liste*<sub>3</sub> est exécutée, et si sa valeur de retour est 0, *liste*<sub>4</sub> est exécutée.

Pour finir, la *liste* suivant l'éventuel **else** de la fin n'est exécutée que si toutes les *listes* des **if/elif** qui précèdent se sont terminées avec une valeur de retour différente de 0.

 La valeur de retour d'un **if** est celle de la *liste* **then** ou **else** qui a été exécutée. S'il n'y en a aucune (ce qui ne se produit que s'il n'y a pas de **else**), la valeur de retour est 0.

## Exemples

Supposons que *monscript* contienne le programme suivant :

```
#!/bin/bash

if (($# != 1)); then
    echo "un argument et un seul svp"
    exit 1
elif (($1 < 0)); then
    echo "argument négatif"
elif (($1 > 0)); then
    echo "argument positif"
else
    echo "argument égal à 0 ou non numérique"
fi
```

Les quelques exemples d'exécution suivants illustrent le rôle du **if** :

```
$ ./monscript
un argument et un seul svp
$ ./monscript -10
argument négatif
$ ./monscript 10
argument positif
$ ./monscript 0
argument égal à 0 ou non numérique
$ ./monscript aaaa
argument égal à 0 ou non numérique
```



## 16.D.2 Les boucles for

Il existe deux type de boucles **for** : l'une que l'on doit à sh et l'autre, propre à bash, qui n'est pas sans rappeler celle d'un langage comme le C (ou C++).


### 16.D.2.a La boucle for de sh

Cette boucle permet d'itérer l'exécution d'une suite d'instructions un nombre de fois fixé.

#### Synopsis

```
for identificateur [ in {mot} ] ; do liste ; done
```

Si la partie **in** {mot} est présente, alors le nombre de mots détermine le nombre d'itérations effectuées. À l'itération *i* la variable *identificateur* prend pour valeur le *i*<sup>e</sup> mot.

 La valeur de retour d'un **for** est la valeur de retour de la *liste* exécutée à la dernière itération. Si aucune itération n'est effectuée, la valeur de retour est 0.

## Exemples

```
$ for var in aaa bbb ccc; do echo "var : $var"; done
var : aaa
var : bbb
var : ccc
$ for var in aaa "un mot" ccc; do echo "var : $var"; done
var : aaa
var : un mot
var : ccc
```

□

Mais cette boucle est bien plus puissante qu'il n'y paraît. En effet, chaque *mot* est sujet à l'interprétation de caractères spéciaux, ce qui permet d'écrire des traitements un peu plus intéressants.

## Exemples

```
$ ls
fic1 fic2 fic3 fic4 fic5 rep1 rep2
$ for var in f*; do echo "fichier commençant par f : $var"; done
fichier commençant par f : fic1
fichier commençant par f : fic2
fichier commençant par f : fic3
fichier commençant par f : fic4
fichier commençant par f : fic5
$ for var in f* *1; do echo "fichier correspondant à f* *1 : $var"; done
fichier correspondant à f* *1 : fic1
fichier correspondant à f* *1 : fic2
fichier correspondant à f* *1 : fic3
fichier correspondant à f* *1 : fic4
fichier correspondant à f* *1 : fic5
fichier correspondant à f* *1 : fic1
fichier correspondant à f* *1 : rep1
$ mots="une variable contenant plusieurs mots"
$ for var in $mots; do echo "var : $var"; done
var : une
var : variable
var : contenant
var : plusieurs
var : mots
```

□

Si la partie **in** {*mot*} est absente, c'est comme si l'on indiquait **in** "\$@" : à l'itération *i*, la variable *identificateur* prendra pour valeur celle du *i*<sup>e</sup> paramètre positionnel.

## Exemple

Supposons que *monscript* contienne le programme suivant :

```
#!/bin/bash

for arg
do
    echo "arg : $arg"
done
```

voici un exemple d'exécution :



```
$ ./monscript arg1 arg2 arg3
arg : arg1
arg : arg2
arg : arg3
$ ./monscript arg1 "un argument" arg3
arg : arg1
arg : un argument
arg : arg3
```

□

Et ce n'est pas tout ! Si un *mot* est sujet à la substitution de variable (ou paramètre), de commande<sup>7</sup> ou d'une expression arithmétique<sup>8</sup>, bash opère une décomposition<sup>9</sup> en mots du résultat de cette substitution. Les mots sont décomposés selon les caractères contenus dans la variable **IFS**, ce qui rend la décomposition paramétrable. Ce mécanisme s'avère très appréciable pour la boucle **for**, comme le montre l'exemple qui suit.

### Exemples

```
$ echo "$PATH"
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11
$ IFS=':'
$ for rep in $PATH; do echo "un rep de PATH : $rep"; done
un rep de PATH : /usr/local/bin
un rep de PATH : /usr/bin
un rep de PATH : /bin
un rep de PATH : /usr/bin/X11
```

⇒ La décomposition en mots est opérée sur la substitution de la variable **PATH**, avec **:** (deux-points) comme séparateur (**IFS**). Elle produit 4 mots (les chemins du **PATH**), ce qui donne 4 itérations de la boucle **for**. À chaque itération, la variable **rep** contient un chemin du **PATH**.

```
$ for rep in $PATH
> do
> echo "rep de PATH : $rep décomposé en :"
> IFS=/
> for ch in $rep
> do
> echo "    $ch"
> done
> done
```

⇒ La chaîne ">\_" en début de ces dernière lignes est le prompt de niveau 2 (contenu dans la variable **PS2**) que bash affiche pour indiquer qu'il attend la fin de l'instruction (ici **for**).

```
rep de PATH : /usr/local/bin décomposé en :
```

```
usr
local
bin
```

```
rep de PATH : /usr/bin décomposé en :
```

```
usr
bin
```

```
rep de PATH : /bin décomposé en :
```

7. Présentée à la section 18.B page 276.

8. Présentée à la section 18.C page 278.

9. Cette décomposition sera détaillée dans la section 18.D page 278.

```
bin
rep de PATH : /usr/bin/X11 décomposé en :
```

```
usr
bin
X11
```

☞ on remarque que la décomposition en mots de `/usr/local/bin` en utilisant comme séparateur `/` donne 4 mots dont le premier est vide. Pour éviter cela, on peut supprimer le premier `/` de `rep` :

```
$ for rep in $PATH
> do
> echo "rep de PATH : $rep décomposé en :"
> IFS=/
> for ch in ${rep#/}
> do
> echo "    $ch"
> done
> done
rep de PATH : /usr/local/bin décomposé en :
usr
local
bin
rep de PATH : /usr/bin décomposé en :
usr
bin
rep de PATH : /bin décomposé en :
bin
rep de PATH : /usr/bin/X11 décomposé en :
usr
bin
X11
```

□

### 16.D.2.b La boucle for des langages évolués

Cette boucle est propre à bash et facilite grandement la vie du programmeur bash car elle est similaire à la boucle `for` du C/C++ (seule la syntaxe est différente).

#### Synopsis

```
for ((expr1 ; expr2 ; expr3)) ; do liste ; done
```

où *expr*<sub>1</sub>, *expr*<sub>2</sub> et *expr*<sub>3</sub> sont des expressions arithmétiques (comme dans **let**, voir section 15.A.1 page 207), dans lesquelles il n'est pas nécessaire de protéger les caractères spéciaux.

Pour cette boucle, *expr*<sub>1</sub> est d'abord évaluée et sert d'initialisation. *expr*<sub>2</sub> est la condition de poursuite de la boucle : *expr*<sub>2</sub> est évaluée et, si elle est vraie, *liste* est exécutée puis *expr*<sub>3</sub> est évaluée, et ce processus recommence à partir de l'évaluation de *expr*<sub>2</sub>. Si l'une de ces trois expressions est manquante, elle est considérée comme vraie.

☞ La valeur de retour de cette boucle **for** est celle de la dernière *liste* exécutée. Si aucune itération n'est faite, la valeur de retour est 0, à moins qu'une expression soit invalide, auquel cas c'est 1.

**Exemple**

```
$ for ((i = 0; i < 5; i++)); do echo "$i"; done
0
1
2
3
4
$ echo $i
5
```


**16.D.3 La boucle while**

La boucle **while** permet d'exécuter une suite de commandes, tant qu'une autre suite de commandes retourne une valeur de retour à 0.

**Synopsis**

```
while liste1; do liste2; done
```

*liste<sub>1</sub>* est la condition de maintien dans la boucle. Elle est d'abord exécutée. Si elle retourne une valeur de retour à 0, alors *liste<sub>2</sub>* est exécutée, et ce processus recommence à partir de l'exécution de *liste<sub>1</sub>*, jusqu'à ce qu'elle retourne une valeur différente de 0.

 La valeur de retour d'une boucle **while** est la valeur retournée par la dernière *liste<sub>2</sub>* exécutée, ou 0 si elle ne l'a jamais été.

**Exemple**

Cet exemple utilise la variable (d'environnement) **PWD** qui contient le chemin absolu du répertoire de travail. On utilise **while** pour compter le nombre de **cd ..** qu'il faut exécuter pour aller à la racine :

```
$ pwd
/users/prof/cpb/public
$ i=0
$ while [ $PWD != / ]; do cd ..; ((i++)); echo "remonte"; done
remonte
remonte
remonte
remonte
$ echo $i
4
```




## 16.D.4 La boucle until

C'est la boucle inverse de **while** : **until** permet d'exécuter une suite de commandes, tant qu'une autre suite de commandes retourne une valeur de retour **différente de 0**.

### Synopsis

```
until liste1 ; do liste2 ; done
```

*liste<sub>1</sub>* est la négation de la condition de maintien dans la boucle. Elle est d'abord exécutée. Si elle retourne une valeur de retour différente de 0, alors *liste<sub>2</sub>* est exécutée, et ce processus recommence à partir de l'exécution de *liste<sub>1</sub>*, jusqu'à ce qu'elle retourne 0.

 La valeur de retour d'une boucle **until** est la valeur retournée par la dernière *liste<sub>2</sub>* exécutée, ou 0 si elle ne l'a jamais été.

### Exemple

Une autre formulation de l'exemple précédent :

```
$ pwd
/users/prof/cpb/public
$ i=0
$ until [ $PWD == / ]; do cd ..; ((i++)); echo "remonte"; done
remonte
remonte
remonte
remonte
$ echo $i
4
```

□

## 16.D.5 Instructions de court-circuitage du déroulement d'une boucle

### 16.D.5.a L'instruction break pour sortir d'une boucle

Bien qu'il soit généralement déconseillé d'arrêter prématurément une boucle, il est des cas où cela s'avère pratique. C'est le rôle de l'instruction **break**.

### Synopsis

```
break [n]
```

Si *n* est omis, il vaut 1 par défaut et **break** sort de la boucle (**for**, **while**, **until** ou **select**) englobante la plus profonde. La valeur de *n* indique le nombre de boucles (les plus profondes) à arrêter : 1 est la boucle englobante la plus profonde, 2 la boucle qui englobe la boucle la plus profonde, etc.

### Exemple

Soit le fragment de programme suivant :

```

for ...
do
  while ...
  do
    if ...
    then break
    fi
    ...
    if ...
    then break 2
    fi
  done
  inst-fin-while
done
inst-fin-for

```

⇒ le premier **break** termine le **while** et *inst-fin-while* est exécutée, alors que le second **break** termine le **for** et *inst-fin-for* est exécutée.

□

### 16.D.5.b L'instruction continue pour passer à l'itération suivante

À l'instar de **break**, l'instruction **continue** court-circuite le déroulement normal d'une boucle.

#### Synopsis

**continue** [*n*]

Si *n* est omis, il vaut 1 par défaut et **continue** fait passer directement à l'itération suivante de la boucle (**for**, **while**, **until** ou **select**) englobante la plus profonde. La valeur de *n* indique le niveau de boucle concernée.

#### Exemple

Soit le fragment de programme suivant :

```

for ...
do
  while ...
  do
    if ...
    then continue
    fi
    inst1-dans-while
    if ...
    then continue 2
    fi
    inst2-dans-while
  done
  inst-dans-for
done

```

⇒ le premier **continue** fait directement passer à l'itération suivante de la boucle **while** sans exécuter les instructions qui suivent (notamment *inst1-dans-while*), alors que le second **continue** fait directement passer à l'itération suivante de la boucle **for** sans exécuter les instructions qui suivent (notamment *inst2-dans-while* et *inst-dans-for*).

□

## 16.D.6 L'instruction **case** pour les branchements multiples

L'instruction **case** joue le rôle de l'instruction *switch* du C/C++, mais ne se limite pas à la comparaison de caractères ou d'entiers.

### Synopsis

```
case mot in { motif { | motif } ) liste ; ; } esac
```

où chaque *motif* { | *motif* } ) *liste* ; ; est appelé un **cas**. Il peut y en avoir un nombre quelconque.

Sa syntaxe se comprend mieux si on l'écrit sur plusieurs lignes avec plusieurs cas et motifs :


```
case mot in
  motif1 )
    liste1
    ; ;
  motif2a | motif2b | motif2c )
    liste2
    ; ;
  motif3a | motif3b )
    liste3
    ; ;
  ...
esac
```


**case** cherche un cas correspondant au *mot*, en commençant par le premier et en les testant dans l'ordre. Seule la *liste* du premier cas correspondant à *mot* est exécutée. C'est une différence notable avec le *switch* du C/C++ car ici le ; ; est obligatoire et joue le rôle du *break* de C/C++.


Avant de chercher sa correspondance dans les différents cas, *mot* est d'abord transformé s'il contient des caractères spéciaux commandant la substitution de variable (ou paramètre), de commande ou d'expression arithmétique. Le tilde est aussi interprété.

Ensuite, le *mot* transformé est comparé aux *motifs* des différents cas. Les *motifs* sont construits avec les mêmes caractères que pour les motifs des noms de fichiers (sans les accolades). Le traitement particulier des / et du . commençant un nom de fichier n'a pas cours dans ce contexte. Pour un cas donné, on peut spécifier plusieurs *motifs* en les séparant par un |. Si l'un des *motifs* du cas correspond au mot (transformé), alors le cas est le seul retenu et sa *liste* est exécutée.

Notons qu'avant que le *mot* (transformé) ne soit comparé aux *motifs*, ceux-ci aussi sont transformés s'il contiennent des caractères spéciaux commandant la substitution de variable (ou paramètre), de commande ou d'expression arithmétique. Le tilde est aussi interprété.

 La valeur de retour d'un **case** est 0 si aucun cas ne s'applique. Sinon, c'est la valeur de retour de la *liste* exécutée.

 Par défaut, la comparaison entre un *motif* et le *mot* est sensible à la casse. On peut la rendre insensible à la casse en activant l'option *nocasematch* en tapant **shopt -s nocasematch**.

 Il n'y a pas de cas *default* : il suffit d'ajouter un cas avec **\*** comme *motif*.

## Exemple

Supposons que *monscript* contienne le programme suivant :

```
#!/bin/bash

case $1 in
  t[oa]*)
    echo "$1 commence par to ou par ta"
    ;;

  *[oa])
    echo "$1 finit par o ou par a"
    ;;

  *)
    echo "$1 ne commence pas par to ou ta"
    echo "et ne finit pas par o ou a"
    ;;
esac
```

Les exemples suivants illustrent les branchements réalisés par **case** :

```
$ ./monscript toto
toto commence par to ou par ta
$ ./monscript tito
tito finit par o ou par a
$ ./monscript titi
titi ne commence pas par to ou ta
et ne finit pas par o ou a
```

□

## 16.D.7 L'instruction select pour créer des menus

L'instruction **select** est en principe destinée à interagir avec l'utilisateur : c'est une boucle qui propose un menu demandant à l'utilisateur de choisir parmi une liste de choix.

### Synopsis

```
select identificateur [ in {mot} ] ; do liste ; done
```


**select** affiche **sur la sortie d'erreur** un menu numéroté à partir de 1, avec une ligne par *mot* indiqué après **in**. Puis, **select** affiche **sur la sortie d'erreur** le contenu de la variable **PS3** (si elle n'existe pas, affiche **#?**) et attend que l'utilisateur entre un choix :

- s'il tape **CTRL-D**, **select** s'arrête ;
- s'il tape une ligne vide, le menu est de nouveau affiché ;
- s'il tape un choix parmi les numéros proposés, le *mot* correspondant est affecté à la variable *identificateur* ;
- s'il tape un choix invalide, *identificateur* prend la valeur vide.

Que le choix soit valide ou invalide, il est affecté à la variable **REPLY**. La *liste* est exécutée pour chaque choix opéré, et **PS3** est de nouveau affiché pour demander à l'utilisateur un nouveau choix. Pour arrêter **select**, il faut utiliser l'instruction **break**.

Comme pour la boucle **for**, si un *mot* est sujet à la substitution de variable (ou paramètre), de commande ou d'une expression arithmétique, bash découpe le résultat de cette substitution en mots, selon les caractères contenus dans la variable **IFS**.

Si **in** {*mot*} n'est pas indiqué, cela revient à écrire **in "\$@"** : les paramètres positionnels sont utilisés comme *mots*.

 La valeur de retour de **select** est la valeur de la dernière *liste* exécutée, ou 0 si aucune ne l'a été.

## Exemples

### Utilisation simple de select

Supposons que *monscript* contienne le programme suivant :

```
#!/bin/bash

PS3="Quelle couleur choisissez-vous ? "
select choix in rouge vert bleu quitter
do
    if [ $choix ]; then
        if [ $choix == quitter ] ; then
            echo "Au revoir"
            break
        fi
        echo "Vous avez choisi le $choix (numéro $REPLY)"
    else
        echo "$REPLY n'est pas un choix proposé"
    fi
done
```

L'exécution suivante montre ce que donnent différentes saisies :

```
$ ./monscript
1) rouge
2) vert
3) bleu
4) quitter
Quelle couleur choisissez-vous ? 3
Vous avez choisi le bleu (numéro 3)
Quelle couleur choisissez-vous ? 5
5 n'est pas un choix proposé
Quelle couleur choisissez-vous ? Entrée
1) rouge
2) vert
3) bleu
4) quitter
Quelle couleur choisissez-vous ? 1
Vous avez choisi le rouge (numéro 1)
Quelle couleur choisissez-vous ? 4
Au revoir
```

### Substitution de tableau dans in

Plutôt que de placer les choix directement dans la partie **in** de **select**, on peut rendre le programme plus générique en utilisant un tableau, qui pourra être réutilisé. Supposons que *monscript2* contienne le programme suivant :



```
#!/bin/bash

PS3="Quelle couleur choisissez-vous ? "
tabchoix=(rouge vert bleu quitter)
select choix in "${tabchoix[@]}"
do
    if [ $choix ]; then
        if [ $choix == quitter ] ; then
            echo "Au revoir"
            break
        fi

        echo "Vous avez choisi le $choix (numéro $REPLY)"
    else
        echo "$REPLY n'est pas un choix proposé"
    fi
done
```

*Voici un exemple d'exécution :*

```
$ ./monscript2
1) rouge
2) vert
3) bleu
4) quitter
Quelle couleur choisissez-vous ? 2
Vous avez choisi le vert (numéro 2)
Quelle couleur choisissez-vous ? 4
Au revoir
```

### *Décomposition en mots dans le in*

*L'exemple qui suit illustre la décomposition en mots selon la variable **IFS**. Le script propose d'écrire les informations détaillées sur un chemin contenu dans la variable **PATH**. Supposons que *monscript3* contienne le programme suivant :*

```
#!/bin/bash

echo "Informations sur les chemins du PATH"

PS3="Quel chemin vous intéresse ? "
OLD_IFS="$IFS"                # sauvegarde de IFS
IFS=":"

select chemin in $PATH quitter
do
    IFS="$OLD_IFS"            # restitution de IFS
    if [ $chemin ]; then
        if [ $chemin == quitter ] ; then
            echo "Au revoir"
            break
        fi

        echo "Voici les informations détaillées sur $chemin :"
        ls -ld $chemin
    else
        echo "$REPLY n'est pas un choix proposé"
    fi
done
```

*Voici un exemple d'exécution :*

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/users/prof/cpb/bin:.
$ ./monscript3
Informations sur les chemins du PATH
1) /usr/local/bin      3) /bin      5) .
```

```

2) /usr/bin                4) /users/prof/cpb/bin  6) quitter
Quel chemin vous intéresse ? 3
Voici les informations détaillées de /bin :
drwxr-xr-x 2 root root 4096 fév  6 10:46 /bin
Quel chemin vous intéresse ? 6
Au revoir

```



## 16.E Gestion des entrées/sorties d'un script

### 16.E.1 Écriture de messages d'erreur

Comme tous les utilitaires, les scripts sont censés écrire les messages d'erreur ainsi que les messages d'invite (messages invitant l'utilisateur à saisir quelque chose) sur la sortie d'erreur. Or, il n'existe pas de commande de type **echo** qui écrive par défaut sur la sortie d'erreur. Même **echo** ne propose pas d'option lui demandant d'écrire sur sa sortie d'erreur.

La solution est alors d'utiliser une commande comme **echo** et de rediriger sa sortie standard sur sa sortie d'erreur en utilisant la redirection **>&2** (contraction de **1>&2**) qui veut dire que la sortie standard devient la même que la sortie d'erreur<sup>10</sup>. Ainsi,

```
echo erreur-ou-invite >&2
```

écrira *erreur-ou-invite* sur la sortie d'erreur du script.

### 16.E.2 read pour lire des entrées

La commande interne **read** permet par défaut de lire une ligne sur son entrée standard, et d'affecter les mots qu'elle contient à une ou plusieurs variables.

#### Synopsis

```
read [-es] [-p invite] [-n nombre] [-d délim] [-a tableau] {identificateur}
```

Sans option, **read** lit une ligne sur l'entrée standard et place chaque mot de la ligne lue dans les variables *identificateur* : le premier dans la première variable, le second dans la deuxième, etc. S'il y a plus de mots que d'*identificateurs*, la dernière variable contiendra le reste de la ligne. S'il y a moins de mots que de variables, les variables sans correspondance de mot auront une valeur nulle.

Si l'option **-a** est utilisée, le *tableau* est créé (écrasé) pour contenir les mots et les *identificateurs* sont ignorés (non modifiés).

En l'absence d'*identificateur* et de l'option **-a**, c'est la variable **REPLY** qui contiendra la ligne lue.

Les délimiteurs des mots de la ligne lue sont ceux contenus dans la variable **IFS** (contenant par défaut espace, tabulation et retour à la ligne).

L'option **-p** remplace avantageusement l'affichage d'une invite avec **echo**. Elle n'est utilisée que si l'entrée se fait à partir du clavier et écrit l'*invite* sur la sortie d'erreur avant de lire une ligne. *invite* est un seul mot (s'il

10. On avait déjà rencontré les redirections *n>&m* dans la section 5.C.4 page 70 pour rediriger les sorties ensemble.


contient des blancs ou autres caractères spéciaux, il faut les protéger).

L'option **-e** provoque une saisie plus agréable pour l'utilisateur car cela lui permet notamment d'utiliser les flèches pour corriger sa saisie.

L'option **-s** demande de ne pas afficher les caractères tapés si la lecture se fait à partir du terminal (clavier). Elle est particulièrement utile pour lire un mot de passe...

L'option **-n** demande d'arrêter la lecture au bout de *nombre* caractères lus plutôt qu'une ligne entière, le retour à la ligne devenant un caractère comme un autre.

L'option **-d** demande de ne plus utiliser le retour à la ligne comme marqueur de fin de ligne mais d'utiliser à la place le caractère *délim* (en fait le premier caractère de *délim*), le retour à la ligne devenant un caractère comme un autre.

 La valeur de retour de **read** est 0 si une ligne est lue (même vide), et 1 si une fin de fichier est lue (au clavier, elle est simulée par **Ctrl-d**).

## Exemples

### Utilisation normale pour la saisie

Supposons que *monscript* contienne le programme suivant :

```
#!/bin/bash

while read -p "que voulez-vous taper ? " mot1 mot2 reste ;
do
    echo "mot1 : $mot1"
    echo "mot2 : $mot2"
    echo "reste : $reste"
done
echo bye
```

Voici quelques exemples d'exécution :

```
$ ./monscript
que voulez-vous taper ? un deux trois quatre
mot1 : un
mot2 : deux
reste : trois quatre
que voulez-vous taper ? toto titi
mot1 : toto
mot2 : titi
reste :
que voulez-vous taper ? Ctrl-d
bye
$ cat fic
un deux trois quatre
toto titi
$ ./monscript < fic
mot1 : un
mot2 : deux
reste : trois quatre
```

```
mot1  : toto
mot2  : titi
reste :
bye
```

⇒ on voit ici que puisque l'entrée n'est pas le clavier, l'invite n'est pas affichée !

### Utilisation de l'option -s (suppression d'echo durant la saisie)

On va illustrer l'option **-s** en demandant un mot de passe afin de poursuivre un programme. Soit *monscript2* contenant le programme suivant :

```
#!/bin/bash

read -s -p 'Mot de passe ? ' lepass
echo                                # pour aller à la ligne

if [ "$lepass" != "superpass" ] ; then # vérification
    echo "échec authentification" >&2
    exit 1
fi

echo "..."                        # suite du traitement
```

Voici quelques exemples d'exécution :

```
$ ./monscript2
Mot de passe ? toto 
échec authentification
```

⇒ la saisie **toto** n'est en réalité pas affichée

```
$ ./monscript2
Mot de passe ? 
échec authentification
```

⇒ rien n'étant saisi, la variable **lepass** est vide

```
$ ./monscript2
Mot de passe ? superpass 
...
```

⇒ **superpass** (non affiché) est bien accepté

□

## 16.E.3 Redirection des entrées/sorties d'une instruction composée ou d'un bloc

Il arrive parfois que l'on souhaite rediriger les entrées/sorties d'une partie d'un script seulement. Par exemple, supposons que l'on demande à l'utilisateur d'entrer un nom de fichier et que l'on souhaite ensuite traiter ce fichier ligne par ligne. Le code peut ressembler à :

```
lire_nom_fichier # lecture au clavier
traiter_fichier  # lecture du fichier
```

`lire_nom_fichier` ne nécessitant *a priori* aucune redirection est très facilement réalisée par l'instruction suivante :

```
read -p "nom du fichier à traiter ? " nomfic
```

Quant à `traiter_fichier`, le code devrait ressembler à :

```
while read ligne
do
    traiter_ligne
done
```

sauf que la lecture doit se faire à partir du fichier contenu dans la variable **nomfic** et non pas sur l'entrée standard.

En premier lieu, il n'est pas correct de remplacer :

```
while read ligne
```

par :

```
while read ligne < $nomfic
```

car dans ce cas, **read ne lirait toujours que la première ligne du fichier !!!** S'il n'est pas vide, on entrerait en plus dans une boucle infinie...

### Solution 1 : déporter le code dans une fonction et rediriger l'entrée à l'appel

Lors d'un appel de fonction, on est libre de rediriger ses entrées/sorties comme on le souhaite (ce qui comprend l'utilisation d'un *pipe*). On peut donc placer le code de traitement du fichier dans une fonction et rediriger l'entrée de la fonction lors de son appel.

Le code suivant est donc tout à fait correct :

```
#!/bin/bash

function traiter_fichier {
    while read ligne
    do
        traiter_ligne
    done
}

read -p "nom du fichier à traiter ? " nomfic
traiter_fichier < $nomfic
```

### Solution 2 : rediriger l'entrée dans la définition de la fonction

Cette solution est un peu plus subtile (mais pas meilleure) : elle tient compte du fait que le corps d'une fonction est placé entre accolades : `{ corps-de-la-fonction }`. Or, **les accolades sont des caractères spéciaux qui forment un bloc d'instructions** qui est vu par bash comme une **commande unique** dont on peut rediriger ses entrées/sorties comme on le souhaite.

Le code suivant est donc tout à fait correct :

```
#!/bin/bash

function traiter_fichier {
    while read ligne
    do
        traiter_ligne
    done
} < $nomfic

read -p "nom du fichier à traiter ? " nomfic
traiter_fichier
```

mais moins générique que le précédent car la fonction `traiter_fichier` utilise la variable `nomfic`.

### Solution 3 : former un bloc d'instructions et rediriger son entrée sans utiliser de fonction

Cette solution utilise la possibilité de former un bloc d'instructions partout où une commande est attendue. Puisqu'un bloc peut avoir ses propres entrées/sorties, on n'est plus forcé d'utiliser une fonction.

Le code suivant est donc aussi tout à fait correct :

```
#!/bin/bash

read -p "nom du fichier à traiter ? " nomfic

{
    while read ligne
    do
        traiter_ligne
    done
} < $nomfic
```

### Solution 4 : toujours utiliser un bloc mais dans un pipe


Cette solution est moins efficace que la précédente. Elle est présentée à titre d'exemple pour montrer qu'un bloc peut figurer dans un pipeline. Pour cela, on utilise la commande `cat` pour lire le fichier, que l'on connecte par un pipe au bloc de traitement du fichier.

Le code résultant, toujours correct, est le suivant :

```
#!/bin/bash

read -p "nom du fichier à traiter ? " nomfic

cat $nomfic | {
    while read ligne
    do
        traiter_ligne
    done
}
```

 L'usage du pipe a un effet pervers qui sera discuté dans la section 17.C page 272. Cette solution n'est pas forcément adaptée au traitement recherché.

### Solution 5 : rediriger l'entrée du bloc `while...done`

Bash traite toutes les instructions `if`, `for`, `while`, `until`, `case` et `select` comme des blocs, où la fin du bloc est matérialisée par le mot clé de fermeture `fi` (pour `if`), `done` (pour `for`, `while`, `until` et `select`) et `esac` (pour `case`). On peut donc se passer de la formation d'un bloc par accolades puisque `while...done` forme déjà un bloc.

Finalement, le simple code suivant est encore tout à fait correct :

```
#!/bin/bash

read -p "nom du fichier à traiter ? " nomfic

while read ligne
do
    traiter_ligne
done < $nomfic
```

### Solution 6 : placer le bloc `while...done` dans un pipeline

Le bloc **while...done** formant une seule instruction, il peut figurer dans un pipeline comme dans le code ci-dessous (moins efficace que le précédent) :

```
#!/bin/bash

read -p "nom du fichier à traiter ? " nomfic

cat $nomfic | while read ligne
do
    traiter_ligne
done
```

 Encore une fois, l'usage du pipe n'est pas forcément approprié (voir section [17.C](#) page [272](#)).

## 16.E.4 Les fichiers virtuels `/dev/stdin`, `/dev/stdout` et `/dev/stderr`


Lorsque, dans un script ou une fonction, bash rencontre une redirection vers les **fichiers virtuels** `/dev/stdin`, `/dev/stdout` ou `/dev/stderr`, alors bash traite ces fichiers de la manière suivante :

- `/dev/stdin` est la même chose que l'entrée standard (descripteur 0) ;
- `/dev/stdout` est la même chose que la sortie standard (descripteur 1) ;
- `/dev/stderr` est la même chose que la sortie d'erreur (descripteur 2).

Ça n'a l'air de rien, mais ces fichiers sont très pratiques. On verra une utilisation de `/dev/stdout` dans la section suivante. Pour l'heure, on dira simplement que pour afficher un message d'erreur ou d'invite, on peut tout aussi bien utiliser la commande :

```
echo erreur-ou-invite >> /dev/stderr
```

qui écrira *erreur-ou-invite* sur la sortie d'erreur du script.

 La raison d'utiliser la redirection `>>` plutôt qu'un simple `>` est que si la sortie d'erreur du script est redirigée dans un fichier et qu'il y a plusieurs messages d'erreur d'écrits, chaque écriture avec `>` écrasera le fichier, ce qui n'est pas le cas avec `>>` ni avec un simple `>&2` !  
On notera néanmoins que d'utiliser `>&2` est plus concis, peut-être un peu moins lisible toutefois.

## 16.F Traitement des options d'un script

Il arrive souvent qu'un script admette des options. Il est d'ailleurs recommandé par la *Free Software Foundation* que tout utilitaire (ou script) admette au moins les options suivantes :

- help** demande au programme d'afficher sur sa sortie standard une brève explication sur son utilité et sur les options reconnues, et se terminer avec un code de retour à 0 en ignorant les autres options ou arguments ;
- version** demande au programme d'afficher sur sa sortie standard son nom, sa version et autres informations utiles, et se terminer avec un code de retour à 0 en ignorant les autres options ou arguments.

La FSF précise sur la page Web <http://www.gnu.org/prep/standards/standards.html#Option-Table> une liste de noms d'options standards qui devraient être utilisés afin de faciliter la vie des utilisateurs. Les noms courts d'options sont aussi possibles comme **-h** pour **--help**. On aura pu remarquer que l'option courte **-v** est généralement utilisée pour **--verbose** (mode "bavard") et non pas pour **--version**. Bien sûr, ces recommandations concernent surtout les programmes destinés à être diffusés.

Du point de vue d'un programme ou d'un script, les options sont passés à travers les paramètres positionnels comme les autres arguments. À charge au script de traiter les paramètres positionnels et de reconnaître les options. Le plus souvent **les options doivent obligatoirement être placées avant les arguments**, bien que certains utilitaires comme **ls** s'accommodent d'un ordre quelconque, mais il vaut mieux ne pas compter sur cette possibilité.

Pour illustrer le traitement des options, on va écrire un script `ndistargs` qui affiche sur sa sortie standard le nombre d'arguments (hors options) distincts et non vides qui lui sont passés. On ne comptera donc qu'une fois un argument qui se retrouve dans plusieurs paramètres positionnels. Ce script devra reconnaître 3 options dont une prenant elle-même un argument :

- **-h** pour afficher son utilisation ;
- **-v** pour un mode bavard, affichant "nombre d'arguments distincts non vides: " avant le nombre calculé ;
- **-o fichier** pour écrire sa sortie dans le fichier *fichier* plutôt que sur la sortie standard.

Avant de commencer l'écriture de `ndistargs`, on va introduire une instruction qui nous sera utile.

### 16.F.1 L'instruction `shift` pour décaler les arguments

Cette instruction peut être utilisée dans un script ou une fonction. Elle a pour effet de modifier les paramètres positionnels du script ou de la fonction qui l'utilise – et seule cette instruction peut le faire<sup>11</sup> – en effectuant un décalage des arguments et en "oubliant" les premiers arguments.

#### Synopsis

**shift** [*n*]

A pour effet de supprimer les *n* premiers arguments et de mettre à jour en conséquence les paramètres positionnels ainsi que les paramètres spéciaux `$#`, `$*` et `$@`. Par défaut, *n* vaut 1 s'il n'est pas spécifié.

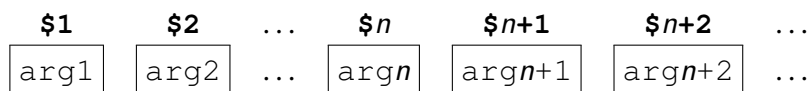
Ainsi, si un script ou une fonction a été appelé(e) de la façon suivante :

```
script-ou-fonction arg1 arg2 ... argn argn+1 argn+2 ...
```

alors, la valeur des paramètres positionnels est la suivante :

<sup>11</sup>. Il y a aussi l'instruction `set` qui le peut mais nous n'avons pas vu comment.





À la suite de l'exécution de **shift n**, les paramètres positionnels deviennent :



et les paramètres spéciaux **\$#**, **\$\*** et **\$@** sont mis à jour.

### Exemple

Supposons que *monscript* contienne le texte suivant :

```
#!/bin/bash

echo "Au début du script :"
echo '$1 contient :' $1
echo '$2 contient :' $2
echo '$3 contient :' $3
echo '$4 contient :' $4
echo '$# contient :' $#
echo '$* contient :' $*

shift

echo -e "\nA la suite de shift :"
echo '$1 contient :' $1
echo '$2 contient :' $2
echo '$3 contient :' $3
echo '$4 contient :' $4
echo '$# contient :' $#
echo '$* contient :' $*

shift 2

echo -e "\nA la suite de shift 2 :"
echo '$1 contient :' $1
echo '$2 contient :' $2
echo '$3 contient :' $3
echo '$4 contient :' $4
echo '$# contient :' $#
echo '$* contient :' $*
```

voici un exemple d'exécution :

```
$ ./monscript arg1 arg2 arg3 arg4
```

Au début du script :

```
$1 contient : arg1
$2 contient : arg2
$3 contient : arg3
$4 contient : arg4
$# contient : 4
$* contient : arg1 arg2 arg3 arg4
```

A la suite de shift :

```
$1 contient : arg2
$2 contient : arg3
$3 contient : arg4
$4 contient :
$# contient : 3
$* contient : arg2 arg3 arg4
```

```
A la suite de shift 2 :  
$1 contient : arg4  
$2 contient :  
$3 contient :  
$4 contient :  
$# contient : 1  
$* contient : arg4
```



## 16.F.2 Exemple de script : ndistargs

Le script **ndistargs** est présenté dans la figure 16.1 pour illustrer quelques notions vues précédemment. Il est découpé en trois parties :

- la première traite les options éventuelles et s'arrête au premier argument ne commençant pas par **-**. Le traitement d'une option est suivi de sa suppression de la liste d'arguments en utilisant **shift**. Une option non reconnue provoque l'affichage d'un message d'erreur ;
- la seconde partie forme un tableau avec les arguments restants et le parcourt afin d'éliminer les doublons ;
- la troisième partie ne fait qu'écrire le résultat, éventuellement en mode bavard et éventuellement dans un fichier<sup>12</sup>.

❗ Tout le calcul peut être remplacé par la simple ligne suivante :

```
res=$(IFS=$'\n' ; echo -e "$*" | grep -v '^$' | sort | uniq | wc -l)
```

utilisant la substitution de commande et la décomposition en mots, présentés au chapitre 18.

❗ Il existe la commande interne de bash appelée **getopts** dont le rôle est de reconnaître les options d'un script. Son utilisation simplifie le traitement et permet de grouper les options (par exemple **-vo** pour **-v -o**), mais elle ne sera pas présentée dans le document. Notons qu'il existe aussi une commande externe appelée **getopt** qui joue le même rôle mais qui en plus peut reconnaître les options de nom long (comme **--help**).

12. En toute rigueur, il faudrait traiter un peu différemment l'écriture dans le fichier de sortie...

```
#!/bin/bash

function usage {
    echo "${0##*/} [-h] [-v] [-o fichier] {argument}"
    echo "    affiche le nombre d'arguments distincts non vides"
    echo
    echo "options :"
    echo "    -h          affiche cette aide"
    echo "    -v          active le mode bavard"
    echo "    -o fichier  écrit la sortie dans fichier"
}

sortie=/dev/stdout

# traitement des options

while (($# != 0))
do
    case $1 in
        -h ) usage
              exit 0
              ;;

        -v ) message="nombre d'arguments distincts non vides: "
              shift
              ;;

        -o ) shift
              if (($# != 0))
              then
                  sortie="$1"
                  shift
              else
                  echo "${0##*/}: erreur option -o nécessite un argument" >&2
                  echo "Pour en savoir davantage, faites ${0##*/} -h" >&2
                  exit 1
              fi
              ;;

        -* ) echo "${0##*/}: erreur option $1 non reconnue" >&2
              echo "Pour en savoir davantage, faites ${0##*/} -h" >&2
              exit 1
              ;;

        * ) break
              ;;
    esac
done

# calcul

tab=("$@")
res=0
for ((i = 0; i < ${#tab[*]}; i++))
do
    if [ -z "${tab[i]}" ]
    then
        continue
    fi

    ((++res))

    for ((j = i+1; j < ${#tab[*]}; j++))
    do
        if [ "${tab[i]}" == "${tab[j]}" ]
        then
            tab[j]=" "
        fi
    done
done

# écriture du résultat

echo "$message$res " > "$sortie"
```

FIGURE 16.1 – Script **ndistargs** synthétisant quelques notions de ce chapitre.



# Chapitre 17

## Environnement des processus

---

### 17.A Définition


Un shell bash, comme tout processus, dispose d'un environnement comprenant :

- le répertoire de travail ;
- le masque de création de fichier (**umask**) ;
- des variables ou tableaux ;
- des fonctions ;
- des alias ;
- une entrée, une sortie et une sortie d'erreur, ainsi que d'autres éventuels (descripteurs de) fichiers ouverts ;
- d'autres choses encore qui sortent du cadre de ce cours.

**Un processus ne modifie que son propre environnement.** Par exemple, s'il modifie une variable ou change de répertoire de travail, cela n'a aucun impact sur l'environnement des autres processus.

### 17.B Héritage

Lorsqu'un processus est exécuté par un autre (notamment par un shell), il **hérite de la copie** d'une partie de l'environnement de son père.

 Par copie, on entend que toute modification de cet environnement par le fils n'affecte en aucun cas l'environnement du père.

La partie copiée est la suivante :

- le répertoire de travail ;
- le masque de création de fichier ;
- les variables, tableaux et fonctions d'environnement ;
- les fichiers ouverts, y compris l'entrée, la sortie et la sortie d'erreur, sauf si elles sont redirigés.
- d'autres choses encore qui sortent du cadre de ce cours.

## 17.C Les pipes et l'environnement des processus

Les commandes d'un pipeline

$$commande_1 \mid commande_2 \mid \dots \mid commande_n$$

sont exécutées en parallèle dans des processus distincts. Chacun dispose donc de sa propre copie d'une partie de l'environnement de son père.

Il y a cependant un cas particulier : lorsqu'une *commande<sub>i</sub>* est une commande interne (ou bloc) de bash alors la commande est exécutée dans un sous-shell<sup>1</sup> qui **hérite de la copie totale de l'environnement du shell courant**, y compris les variables locales, les alias, etc.

La modification de ces variables n'affecte que le sous-shell et pas le shell courant. On peut alors critiquer les solutions 4 et 6 que nous avons élaborées pour traiter un fichier (voir section 16.E.3 page 262). Elles utilisent un pipe et sont les suivantes :

### Solution 4 :

```
#!/bin/bash

read -p "nom du fichier à traiter ? " nomfic

cat $nomfic | {
    while read ligne
    do
        traiter_ligne
    done
}
```

### Solution 6 :

```
#!/bin/bash

read -p "nom du fichier à traiter ? " nomfic

cat $nomfic | while read ligne
do
    traiter_ligne
done
```

Supposons que le but du traitement du fichier est de compter le nombre de lignes pour le réutiliser plus tard. On pourrait penser qu'il suffit de modifier les scripts en ajoutant les lignes en gras :

### Solution 4' :

```
#!/bin/bash

read -p "nom du fichier à traiter ? " nomfic
nbl=0
cat $nomfic | {
    while read ligne
    do
        ((nbl++))
        echo "boucle: nbl=$nbl"
    done
}
echo "fin: nbl=$nbl"
```

### Solution 6' :

```
#!/bin/bash

read -p "nom du fichier à traiter ? " nomfic
nbl=0
cat $nomfic | while read ligne
do
    ((nbl++))
    echo "boucle: nbl=$nbl"
done
echo "fin: nbl=$nbl"
```

où l'on affiche dans la boucle la valeur courante de la variable **nbl**. Supposons qu'on dispose d'un fichier **ficlignes** qui contient 3 lignes de texte. Alors l'exécution de ces scripts donne :

### Exécution de la solution 4'

```
$ ./sol4p
nom du fichier à traiter ? ficlignes
boucle: nbl=1
boucle: nbl=2
boucle: nbl=3
fin: nbl=0
```

1. comme si l'on utilisait des parenthèses, voir section 18.A page 275.

*Exécution de la solution 6'*

```
$ ./sol6p
nom du fichier à traiter ? ficlignes
boucle: nbl=1
boucle: nbl=2
boucle: nbl=3
fin: nbl=0
```

On voit que dans les deux cas, on n'obtient pas le résultat attendu. En effet, la variable **nbl** qui évolue dans la boucle est celle du sous-shell. Or, celle qu'on utilise à la fin est celle du shell exécutant le script, qui est restée inchangée !! Ainsi, ces solutions ne sont pas adaptées à ce traitement, mais les autres (1, 2, 3 et 5) le sont.

❶ Nous verrons au chapitre suivant que l'on peut facilement placer dans une variable **nbl** le nombre de lignes d'un fichier obtenu avec la commande **wc** en écrivant la simple ligne suivante :

```
nbl=$(cat $nomfic | wc -l)
```

## 17.D Utilité des variables, tableaux et fonctions d'environnement

Il n'est pas possible de décrire de manière exhaustive toutes les utilisations possibles des variables, tableaux et fonctions d'environnement. On peut dire que c'est une manière simple pour un processus de communiquer des informations à son fils.

Certains utilitaires lorsqu'ils sont exécutés, examinent les variables d'environnement car certaines peuvent les concerner. Par exemple, la commande **ls** regarde si la variable d'environnement **LS\_COLORS** existe et si c'est le cas, colore les noms de fichiers selon les directives contenues dans cette variable.

Bash, lorsqu'il est exécuté non interactivement (notamment pour exécuter un script), regarde si la variable d'environnement **BASH\_ENV** existe et, si c'est le cas, exécute le fichier indiqué par cette variable avant d'exécuter le script.

Enfin, il est courant dans un script d'utiliser des informations telles que le nom et/ou le répertoire d'accueil de l'utilisateur qui l'exécute. Ces informations sont respectivement contenues dans les variables d'environnement **USER** et **HOME**.

## 17.E Création de variables, tableaux et fonctions d'environnement

Il y a plusieurs possibilités. On se limitera à l'utilisation de la commande interne **export**.

### Synopsis

```
export [-nf] { identificateur [=mot] }
```

*identificateur* peut être un identificateur de variable ou de fonction (les tableaux d'environnement ne sont pas encore implémentés).

Sans option, **export** rend d'environnement les variables *identificateurs* qui sont créées si elles n'existent pas. Chaque *identificateur* (de variable) peut être suivi de **=mot** pour lui attribuer *mot* comme valeur (même syntaxe

que la première forme de la création de variable de type chaîne, voir section [15.A.1](#) page [207](#)).

Pour exporter une fonction (qui doit déjà exister), il faut utiliser l’option **-f**. Notons que pour que cette exportation soit utile, il faut que le processus qui en hérite sache exploiter ces fonctions...

Si aucune option ni *identificateur* n’est indiqué, **export** se comporte comme **declare -x** (voir section [15.C](#) page [223](#)) et affiche la liste des variables et tableaux d’environnement.

**export -f** seul (synonyme de **declare -fx**) affiche la liste des fonctions d’environnement.

L’option **-n** fait l’opération inverse : les variables *identificateur* (et les fonctions si **-f**) perdent leur statut d’environnement.



# Chapitre 18

## Autres fonctionnalités de bash

---

### 18.A Création d'un sous-shell en utilisant les parenthèses


Il est possible d'exécuter des commandes dans un sous-shell sans passer par un script. En effet, là où une commande (ou une suite de commandes) est admise, on peut écrire la construction spéciale :

(*commande*)


où *commande* peut être n'importe quelle suite de commandes.

Cette construction a les mêmes caractéristiques qu'un bloc formé par des accolades, **à ceci près que la commande sera exécutée dans un sous-shell**, alors qu'un bloc ne provoque pas de création d'un sous-shell. Tout comme un bloc, le sous-shell créé pour exécuter *commande* dispose de ses propres entrées/sorties qui peuvent être redirigées.

Ce qui distingue un bloc d'un sous-shell est que **le sous-shell dispose de son propre environnement**. À sa création, il hérite d'une copie totale<sup>1</sup> de l'environnement de son père.

 C'est une différence notable avec les processus créés pour exécuter une commande (ou un script) qui n'héritent pas d'une copie des variables locales ou des alias du père (voir section **17.B** page 271).

Disposant de son propre environnement, toute modification du répertoire de travail et toute création/modification de variables (y compris d'environnement), de fonctions ou d'alias opérée dans un sous-shell n'a aucune incidence sur l'environnement du père qui reste inchangé.

 Les sous-shells créés pour la substitution de commande (voir plus loin), ainsi que les commandes lancées en tâche de fond disposent aussi d'une copie totale de l'environnement de leur père.

L'utilisation des sous-shells créés avec les parenthèses est assez marginale et n'a de sens que si on souhaite exécuter du code qui modifie l'environnement, alors que l'on ne veut pas modifier l'environnement du shell courant.

---

1. Modulo certaines choses qui sortent du cadre de ce cours.

## Exemples

```
$ pwd ; (cd /bin; pwd) ; pwd
/home/cyril
/bin
/home/cyril
```

⇒ Le changement de répertoire n'est effectif que pour le sous-shell. Le shell courant n'étant pas affecté par ce changement, le dernier **pwd** donne le même résultat que le premier.

```
$ pwd ; (cd /bin; pwd) > ici ; pwd
/home/cyril
/home/cyril
```

⇒ La sortie du sous-shell est redirigée dans le fichier *ici* (de */home/cyril*).

```
$ tab=(un deux trois)
$ var=blablabla
$ function ecrire { echo "($1) tab: ${tab[*]} et var : $var"; }
$ (ecrire fils) ; ecrire pere
(fils) tab: un deux trois et var: blablabla
(pere) tab: un deux trois et var: blablabla
```

⇒ on voit que le fils "connaît" le tableau **tab**, la variable **var** et la fonction **ecrire**

```
$ (var=xxxx ; ecrire fils) ; ecrire pere
(fils) tab: un deux trois et var: xxxx
(pere) tab: un deux trois et var: blablabla
```

⇒ on voit que la modification de **var** par le fils n'est effective que pour ce dernier

```
$ export IFS
$ (IFS=',' ; var=xxxx ; ecrire fils) ; ecrire pere
(fils) tab: un,deux,trois et var: xxxx
(pere) tab: un deux trois et var: blablabla
```

⇒ on voit que même si **IFS** est rendue d'environnement, sa modification dans le fils n'affecte pas le père.

□

## 18.B Substitution de commande

La substitution de commandes est une des fonctionnalités les plus intéressantes des shells. Elle permet de remplacer n'importe quelle commande (ou suite de commandes) par ce que cette commande écrit sur sa sortie standard. En bash, la substitution de commandes est demandée par la construction spéciale :

**\$ (commande)**

où *commande* peut être n'importe quelle suite de commandes.

✍ Cette substitution se protège comme la substitution de variables : entre quotes ou par un backslash, à ceci près que les parenthèses sont aussi des caractères spéciaux qu'il faut protéger.

Si elle n'est pas protégée, cette construction est remplacée par ce qu'écrit *commande* sur sa sortie standard, où les éventuels derniers retours à la ligne sont supprimés. Les autres retours à la ligne sont gardés mais le traitement

(décomposition en mots) du résultat de la substitution peut éventuellement les traiter comme des espaces.

Si la substitution de commandes n'est pas placée entre guillemets, le résultat de la substitution est sujet à la décomposition en mots et à la substitution des motifs de noms de fichiers.

Les caractères spéciaux placés entre les parenthèses sont traités au niveau de la *commande* et non pas au niveau de la ligne de commandes qui contient la construction.

Enfin, les substitutions de commandes peuvent être imbriquées !

❗ Il existe une forme ancienne de substitution de commande qui utilise les caractères spéciaux *back-quotes* (apostrophes inversées) :

``commande``

Dans certains shells (comme `csh` ou `tcsh`), c'est la seule disponible. Elle est moins pratique à utiliser et surtout à imbriquer et nous ne l'utiliserons pas.

## Exemples

Premier exemple très simple :

```
$ cat lesfics
fic1
fic2
fic3
$ cat $(cat lesfics) > fics
```

⇨ cette dernière commande crée le fichier *fics* contenant la concaténation des fichiers *fic1*, *fic2* et *fic3*. En effet, `$(cat lesfics)` est remplacé par le contenu du fichier *lesfics* qui, après découpage en mots, donne les 3 arguments *fic1*, *fic2* et *fic3* à `cat` dont la sortie est redirigée.

Exemple un peu plus complexe :

```
$ cat $(head -n 1 lesfics ; tail -n 1 lesfics) > fics
```

⇨ place dans *fics* la concaténation des fichiers *fic1* et *fic3*. En effet, `head` n'écrit que la première ligne de *lesfics*, alors que `tail` n'écrit que sa dernière ligne.

Exemple pas forcément plus complexe :

```
$ ls -l
total 20
-rw-r--r-- 1 cyril users 57 2008-03-05 11:38 fic1
-rw-r--r-- 1 cyril users 107 2008-03-05 11:39 fic2
-rw-r--r-- 1 cyril users 80 2008-03-05 11:39 fic3
-rw-r--r-- 1 cyril users 137 2008-03-05 11:52 fics
-rw-r--r-- 1 cyril users 15 2008-03-05 11:50 lesfics
$ infosfic1=$(ls -l fic1)
```

⇨ création d'un tableau `infosfic1` contenant les informations détaillées sur *fic1*

```
$ echo "fic1 contient ${infosfic1[4]} octets"
fic1 contient 57 octets
```

Autres possibilités :

```
$ echo "fic1 contient $(wc -c fic1) octets"
fic1 contient 57 fic1 octets
```

➡ `wc -c fic1` écrit sur sa sortie 57 *fic1*, que l'on retrouve dans la chaîne en argument de `echo`.

On peut mieux faire si on utilise un pipe :

```
$ echo "fic1 contient $(cat fic1 | wc -c) octets"
fic1 contient 57 octets
```

➡ cette fois, `wc` n'a pas connaissance du nom de fichier et ne l'écrit pas.

Exemple dans le même genre mais avec des substitutions imbriquées :

```
$ echo "taille totale des fichiers de lesfics : $(cat $(cat lesfics) | wc -c) octets"
taille totale des fichiers de lesfics : 244 octets
```

□

## 18.C Substitution d'expressions arithmétiques

Cette substitution est demandée par la construction suivante :

`$( (expression) )`

où *expression* est une expression arithmétique comme dans **let** (voir section [15.A.1 page 207](#)).

✍ Cette construction se protège de la même manière que la substitution de commandes.

Si elle n'est pas protégée, elle est remplacée par le résultat de l'évaluation de *expression*.

*expression* peut contenir des substitutions de commandes ou de variables, ainsi que d'autres substitutions d'expressions arithmétiques. Les autres caractères spéciaux qu'elle peut contenir n'ont pas besoin d'être protégés.

### Exemples

```
$ echo "2 * 3 = $( (2 * 3) )"
2 * 3 = 6
$ echo "2 * taille de fic1 = $( (2 * $(cat fic1 | wc -c)) )"
2 * taille de fic1 = 114
```

□

## 18.D Décomposition en mots

Si des substitutions de variables (ou paramètres), de commandes ou d'expressions arithmétiques sont placées entre guillemets, alors leur résultat forme toujours un seul mot, à part pour les substitutions de tableaux ou paramètres positionnels utilisant `@` qui forment plusieurs mots.

Si elles ne sont pas placées entre guillemets, bash décompose leur résultat en mots en utilisant les caractères contenus dans la variable **IFS** comme séparateurs de mots. Si cette variable n'existe pas, sa valeur par défaut est

utilisée : espace, tabulation, retour à la ligne. Si elle existe mais ne contient pas sa valeur par défaut, les caractères qu'elle contient sont utilisés comme séparateurs de mots. Si elle contient un espace, les tabulations servent aussi de séparateurs de mots. Si elle existe mais est vide, les mots ne sont pas décomposés.

❗ Par défaut, la variable **IFS** est créée avec l'instruction :

```
IFS=$' _\t\n'
```

où la construction spéciale `$' chaîne '` s'appelle le *ANSI-C Quoting* (voir [http://www.gnu.org/software/bash/manual/bashref.html#ANSI\\_002dC-Quoting](http://www.gnu.org/software/bash/manual/bashref.html#ANSI_002dC-Quoting)). Son rôle est de remplacer certaines séquences spéciales par le caractère correspondant en ANSI C standard.

Il est impératif de l'utiliser<sup>2</sup> pour placer dans **IFS** des caractères tels que la tabulation (`\t`) ou le retour à la ligne (`\n`).

## Exemples

Pour illustrer ce concept, nous allons utiliser la boucle **for** pour traiter un à un les mots obtenus par la décomposition en mots du résultat de la substitution d'une variable. Au départ, **IFS** vaut `$' _\t\n'` :

```
$ texte="tres__petite__phrase"
$ for var in $texte; do echo "iteration avec var=$var"; done
iteration avec var=tres
iteration avec var=petite
iteration avec var=phrase
```

⇒ on a bien 3 mots. On constate que deux mots peuvent être séparés par un nombre quelconque (supérieur à 1) de caractères de **IFS**

```
$ OIFS="$IFS"
```

⇒ pour une restitution ultérieure

```
$ IFS="e"
```

⇒ à l'avenir, seul le **e** sera utilisé comme séparateur de mots dans la décomposition

```
$ for var in $texte; do echo "iteration avec var=$var"; done
iteration avec var=tr
iteration avec var=s__p
iteration avec var=tit
iteration avec var=__phras
```

⇒ les **e** séparent 4 mots comme suit `tr` `e` `s__p` `e` `tit` `e` `__phras` `e`, où les espaces sont des caractères comme d'autres

```
$ IFS="e_"
$ for var in $texte; do echo "iteration avec var=$var"; done
iteration avec var=tr
iteration avec var=s
iteration avec var=p
iteration avec var=tit
iteration avec var=phras
```

2. Et en ce qui me concerne, c'est le seul moment où je l'utilise.

⇨ en se servant des **e** et des blancs comme séparateurs, on obtient 5 mots :

`tr e s p e tit e phras e`

\$ **IFS="\$OIFS"**

⇨ restitution de la valeur de **IFS**

□

## 18.E Traitement de la ligne de commandes

De nombreuses substitutions étant possibles, il est temps d'indiquer dans quel ordre elles sont réalisées. Il est difficile de décrire précisément tous les traitements d'une ligne de commandes mais, dans les grandes lignes, ils sont opérés dans cet ordre :

### 1. Décomposition en jetons

La ligne de commande est décomposée en jetons séparés par les caractères espace, tabulation, retour à la ligne, ; , ( , ) , < , > , | et &

### 2. Traitement du premier jeton

S'il n'est pas protégé par un backslash ou une apostrophe, bash regarde s'il s'agit d'un mot clé d'ouverture (comme **if**, **for**, etc.) ou tel que **function**, **{** ou **(** qui marquent le début d'une commande composée. Si c'est le cas, bash passe au mot suivant, recommence à partir de 1 tout en gardant en mémoire qu'il faudra un mot clé ou un symbole de fermeture. S'il s'agit d'un mot clé mal placé, bash signale une erreur

### 3. Test d'alias

Si le premier jeton est un alias existant, bash le remplace par le contenu de l'alias et recommence à partir de 1

### 4. Expansion des accolades (voir section 4.E page 58)

### 5. Expansion du tilde (voir section 4.C page 55)

### 6. Substitution des variables et des paramètres (voir chapitre 15 page 207)

Le résultat de ces substitutions n'est pas soumis à la substitution des commandes ni des expressions arithmétiques

### 7. Substitution des commandes

Le résultat de ces substitutions n'est pas soumis à la substitution des expressions arithmétiques

### 8. Substitution des expressions arithmétiques

### 9. Décomposition en mots

Ne touche que le résultat des substitutions 6, 7 et 8 non placées entre guillemets

### 10. Expansion des chemins de fichiers (voir section 4.D page 56)

### 11. Redirections et exécution de la commande

Si une substitution (de variable, commande ou expression arithmétique) est encadrée par des guillemets, elle n'est pas soumise à la décomposition en mots ni à l'expansion des chemins de fichiers.

Les guillemets protègent aussi de la substitution des accolades et du tilde, et les quotes protègent de tous les traitements.

# Annexe A

## L'éditeur vi

La commande externe **vi** est, avec **emacs**, un éditeur de texte presque aussi vieux que le système Unix. Il est depuis longtemps livré avec toutes les distributions d'Unix. Il est certainement encore aujourd'hui l'un des éditeurs de texte les plus puissants, bien qu'il puisse sembler très archaïque.

❗ **vi** n'opère qu'en mode texte, c'est à dire qu'il n'affiche aucun menu déroulant, sur lequel on pourrait cliquer avec la souris. Toutes les commandes qu'il reconnaît sont uniquement déclenchées avec le clavier. Il fait partie des connaissances incontournables d'un informaticien car il arrive que le mode graphique ne soit pas opérationnel, alors qu'on a besoin d'éditer un fichier.

Au prix d'un investissement minimum (certes un peu fastidieux) dans l'apprentissage de ses commandes, **vi** est un éditeur très performant. La version de **vi** que nous utilisons est en réalité une version améliorée qui s'appelle **vim** (pour *vi improved*). Elle apporte d'autres fonctionnalités par rapport à **vi**.

### Synopsis

**vi** [-r] [+n] [référence]

*référence* (si spécifiée) est une référence à un fichier existant ou à un fichier qui sera créé. L'option **-r** (*recover* ou restauration) permet de restaurer un fichier à la suite d'un problème survenu pendant son édition avec **vi** (arrêt brutal du système, ...). L'option **+n** demande de se positionner à la ligne *n* du fichier. C'est une option utile lorsqu'on veut corriger une ligne d'un programme signalée incorrecte par un compilateur. Bien d'autres options sont reconnues par **vi** (et par **vim**).

❗ Bien que volumineux, ce chapitre ne présente que les possibilités les plus couramment utilisées. Bien d'autres existent. N'hésitez pas à consulter le manuel en ligne, l'aide de **vi**, ou même le site officiel de **vim** : <http://www.vim.org>.

## A.1 Fonctionnement

Lorsque **vi** est lancé, le fichier (nouveau ou ancien) est affiché. On se trouve alors en **mode commandes**. Ce mode permet de sauver le fichier, d'en lire un autre, mais aussi d'en copier- coller une partie ou de la supprimer, de rechercher du texte, d'en remplacer, etc. Mais il ne permet pas d'insérer du texte tapé au clavier. Pour cela, il faut être en **mode insertion** ou en **mode remplacement**.

En mode commandes, la plupart des commandes de l'éditeur sont tapées « **en aveugle** », c'est à dire qu'on ne voit pas ce que l'on tape. Seules les commandes commençant par **/**, **?** ou **:** sont affichées en bas de l'écran.

En mode insertion ou en mode remplacement, le texte tapé est affiché, jusqu'à ce que l'on sorte du mode insertion ou remplacement (pour revenir au mode commandes).

Il existe aussi le **mode visuel** qui permet d'appliquer des commandes à du texte sélectionné.

## A.2 Entrée dans le mode insertion

Pour entrer (simplement) dans le mode insertion à partir du mode commandes, il suffit de taper une des lettres suivantes :

- i** insérer à gauche de la position courante du curseur
- I** insérer à gauche du premier caractère de la ligne courante
- a** insérer à droite de la position courante du curseur
- A** insérer à droite du dernier caractère de la ligne courante
- o** insérer une nouvelle ligne au-dessous de la ligne courante
- O** insérer une nouvelle ligne au-dessus de la ligne courante

### Entrée en supprimant du texte

- [n] s** supprime *n* caractères puis passe en mode insertion
- [n] cw** supprime *n* mots à partir du curseur, puis passe en mode insertion
- [n] cc** supprime *n* lignes, puis passe en insertion pour la nouvelle ligne

**i** Par défaut, *n* vaut 1 (s'il n'est pas spécifié). C'est le cas pour toute commande pouvant être préfixée par un nombre.

## A.3 Entrée dans le mode remplacement

Elle se fait en tapant **R** dans le mode commandes. Dans le mode remplacement, tout ce qui est tapé remplace le texte sous le curseur jusqu'à ce que l'on sorte du mode remplacement (pour revenir au mode commandes).

## A.4 Sortie du mode insertion et du mode remplacement

Pour revenir au mode commandes, il suffit de taper sur la touche ESC (*Escape* ou *Échappement*).

## A.5 Mode commandes

En mode commandes, de nombreuses commandes sont disponibles. Certaines sont indiquées ici, et regroupées par thèmes. La description d'autres commandes peut être obtenue dans l'aide de **vi**, disponible à partir du mode commandes en tapant :

**:help** [commande]



## A.5.1 Commandes édition de fichier

Les séquences de caractères suivantes permettent de contrôler l'édition du fichier :


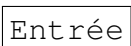





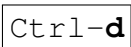
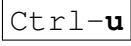
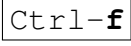
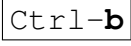
- : [étendue] **w** [!] [nom] sauvegarde le fichier sous le nom *nom*, sans sortir de **vi**. Le *nom* ne doit être spécifié que si l'on veut sauvegarder le fichier en cours d'édition sous un nom différent de celui qu'il porte. Il est possible de ne sauvegarder que la partie du fichier spécifiée par *étendue*. Voir « *Substitution de chaînes* » (section A.5.6 page 285) pour une description de *étendue*. Le caractère optionnel **!** force l'écrasement du fichier *nom* s'il existe, ou force l'enregistrement sur un fichier sur lequel on n'a pas le droit d'écriture, mais qui est contenu dans un répertoire sur lequel on a le droit d'écriture ;
- : **sav** [!] [nom] c'est l'*enregistrer-sous* des applications Windows classiques. Le **!** force l'écrasement du fichier *nom* s'il existe ;
- ZZ** ou : **wq** ou : **x** sauvegarder les modifications et sortir de **vi** ;
- : **e** *nom* passer à l'édition du fichier *nom* ;
- : **r** *nom* insérer le fichier *nom* à la ligne qui suit le curseur ;
- : **q** sortir de **vi** si aucune modification n'a été faite depuis la dernière sauvegarde ;
- : **q!** sortir de **vi** sans sauvegarder les modifications.

Dans la commande précédente le caractère **!** sert à forcer **vi** à réaliser l'opération demandée. En effet, si le fichier avait été modifié, **vi** aurait refusé de quitter sans le sauvegarder. Le **!** l'oblige à quitter sans sauvegarder.

**i** Le **!** peut être utilisé pour les autres commandes d'édition. Par exemple : **e nom !** force à passer à l'édition du fichier *nom* même si le fichier en cours d'édition avait été modifié...

## A.5.2 Déplacements

### Déplacements simples

-  ou **j** se déplacer d'une ligne vers le bas
-  aller au début de la ligne suivante
-  ou **k** se déplacer d'une ligne vers le haut
-  ou **h** ou  se déplacer vers la gauche
-  ou **l** ou  se déplacer vers la droite
-  défilement du fichier d'un demi écran (nombre de lignes de la fenêtre divisé par 2) vers le bas
-  défilement du fichier d'un demi écran vers le haut
-  défilement du fichier d'un écran vers le bas
-  défilement du fichier d'un écran vers le haut

- H** se positionner sur la première ligne de la fenêtre
- M** se positionner sur la ligne au milieu de la fenêtre
- L** se positionner sur la dernière ligne de la fenêtre
- \$** se positionner sur le dernier caractère de la ligne courante
- O** se positionner sur le premier caractère de la ligne courante
- ^** se positionner sur le premier caractère non blanc de la ligne courante
- G** se positionner à la fin du fichier
- :n** se positionner sur la ligne numéro *n*

## Déplacements par recherche

- %** lorsque le curseur est placé sur l'un des caractères « *ouvrants* » [, ( ou {, ou l'un des caractères « *fermants* » ], ) ou }, se placer sur le caractère fermant ou ouvrant correspondant (en respectant les imbrications)
- [n] [{** se place sur le *nième* { (précédent) qui n'a pas de correspondance avec un } (jusqu'au curseur);
- [n] [(** se place sur le *nième* ( (précédent) qui n'a pas de correspondance avec un ) (jusqu'au curseur);
- [n] ]}** se place sur le *nième* } (suivant) qui n'a pas de correspondance avec un { (à partir du curseur);
- [n] ])** se place sur le *nième* ) (suivant) qui n'a pas de correspondance avec un ( (à partir du curseur);
- [n] w** déplacement de *n* mots en avant par rapport à la position courante du curseur
- [n] b** déplacement de *n* mots en arrière par rapport à la position courante du curseur
- [n] e** se placer sur le dernier caractère du *nième* mot suivant
- /expreg** déplacement jusqu'à la prochaine occurrence d'une chaîne correspondant à l'expression régulière *expreg*. Les expressions régulières sont étudiées au chapitre 12. Pour l'instant, on dira que si *expreg* commence par le caractère ^ c'est que la chaîne recherchée doit se trouver en début de ligne. Si *expreg* se termine par \$, c'est qu'elle doit se trouver en fin de ligne.
- ?expreg** déplacement jusqu'à l'occurrence précédente d'une chaîne correspondant à l'expression régulière *expreg*;
- /** refait la dernière recherche, vers la fin du fichier;
- ?** refait la dernière recherche, vers le début du fichier;
- [n] n** refait *n* fois le dernier déplacement

## Exemples de recherches possibles (avec expression régulières)

- /^truc** déplacement jusqu'à la prochaine ligne commençant par **truc**
- /truc\$** déplacement jusqu'à la prochaine ligne se terminant par **truc**
- /[tT]ruc** déplacement jusqu'à la prochaine occurrence de **truc** ou de **Truc**
- ?[tT]ruc** déplacement jusqu'à l'occurrence précédente de **truc** ou de **Truc**

### A.5.3 Effacements

**J** concaténation de la ligne où est le curseur, avec la suivante

[*n*] **x** effacer *n* caractères

[*n*] **dw** effacer *n* mots

**d0** effacer de la position courante du curseur jusqu'au début de la ligne

**d\$** ou **D** effacer de la position courante du curseur jusqu'à la fin de la ligne

[*n*] **dd** effacer *n* lignes

### A.5.4 Remplacement

[*n*] **rc** remplace *n* caractères à partir du curseur par le caractère *c*

### A.5.5 Copier-coller

D'une façon générale, la dernière suppression effectuée sur le texte est gardée en mémoire pour être éventuellement collée au moyen d'une des deux commandes suivantes :

**P** (majuscule) collage à gauche du curseur (ou ligne précédente si collage de ligne)

**p** (minuscule) collage à droite du curseur (ou ligne suivante si collage de ligne)

Mais on peut aussi copier du texte (qui pourra aussi être collé avec **p** ou **P**) au moyen des commandes suivantes :

**y** [*n*] **w** copie *n* mots à partir du curseur

**y** [*n*] **y** ou [*n*] **yy** copie *n* lignes à partir du curseur

: [*étendue*] **y** copie les lignes spécifiées par *étendue* (voir ci-après pour *étendue*)

### A.5.6 Substitution de chaînes

La substitution est demandée par la commande : **s** dont la forme générale est ci-dessous. D'autres possibilités sont offertes. Taper :**help** :**s** dans **vi** pour en savoir plus.

: [*étendue*] **s** / *expreg* / *chaîne* [ / [*options*] ] substitution d'une chaîne correspondant à l'expression régulière *expreg* par *chaîne* sur toutes les lignes précisées par l'*étendue* avec les *options*

#### Explications :

*étendue* est de la forme [*n* [, *m*] ]. Veut dire de la ligne *n* à la ligne *m*. *n* et *m* sont des entiers ou de la forme [ ( . | \$ ) [ ( + | - ) *x* ] ], où :

. indique la ligne courante

\$ indique la dernière ligne

*x* est un entier

Ainsi .+5 veut dire "5 lignes après la ligne courante" et \$-2 veut dire "2 lignes avant la dernière ligne". L'*étendue* ., \$ veut dire de la ligne courante jusqu'à la fin du fichier et l'*étendue* 1, \$ veut dire sur toutes les lignes du fichier ;

**i** Il existe l'étendue % qui est une abréviation pour **1, \$**

**s** substitution ;

**/expreg/chaîne** *expreg* est une expression régulière (sorte de motif) représentant les chaînes à remplacer alors que *chaîne* est la chaîne qui les remplacera. Les expressions régulières sont étudiées au chapitre 12. Pour l'instant, on dira que *expreg* est une simple chaîne de caractères ;

**options** une combinaison des caractères suivants (consulter l'aide de **vi** en tapant **:help :s\_flags** pour toutes les possibilités) :

**g** (*global*) pour toutes les apparitions dans les lignes indiquées. Sans **g**, sur chaque ligne indiquée, seule la première chaîne correspondant à *expreg* est remplacée par *chaîne* ;

**c** (*confirmation*) demande confirmation avant d'opérer une modification. Comme confirmation, il sera possible de répondre oui (**y**), non (**n**), oui pour toute la suite (**a**) et d'autres choses ;

**i** (*ignore casse*) ne pas distinguer les majuscules et les minuscules pour la chaîne à rechercher.

## A.5.7 Commande globale

Il est possible de réaliser certaines commandes sur un ensemble de lignes qui satisfont à certains critères. C'est ce que permet la commande **global** dont la syntaxe est :

: [étendue] **g** / *expreg* / *commande* effectue la *commande* sur toutes les lignes de *étendue* dont une chaîne correspond à *expreg*. La *commande* peut être complexe. Cela peut être aussi simplement **d** pour la suppression des lignes correspondantes.

## A.5.8 Partage de la fenêtre vi

Il est possible de partager la fenêtre **vi** en plusieurs sous-fenêtres indépendantes. Cela permet d'éditer plusieurs fichiers dans la même fenêtre, ou même d'éditer des parties différentes d'un même fichier dans des sous-fenêtres.

### A.5.8.a Création de sous-fenêtres

[*n*] **Ctrl-w s**

ou : [*n*] **sp[lit]** partage la fenêtre où se trouve le curseur en deux sous-fenêtres éditant le même fichier. Si spécifié, *n* indique le nombre de lignes de la nouvelles fenêtre (et prélevées aux lignes de la fenêtre d'origine) ;

[*n*] **Ctrl-w n**

ou : [*n*] **new** même chose mais la nouvelle fenêtre édite un nouveau fichier ;

: [*n*] **sp[lit]** *référence*

ou : [*n*] **new** *référence* même chose mais permet de spécifier le fichier *référence* (existant ou nouveau) à éditer dans la nouvelle fenêtre.

### A.5.8.b Fermeture d'une (sous-)fenêtre

Utiliser `Ctrl-w q` ou les commandes normales `:q`, `:x`, etc.

### A.5.8.c Déplacement entre sous-fenêtres

`Ctrl-w ↑` passer à la fenêtre supérieure

`Ctrl-w ↓` passer à la fenêtre inférieure

## A.5.9 Divers

**u** annule la dernière modification sur la ligne, puis l'avant dernière, etc.

**U** annule les modifications sur la ligne, puis les restaure, puis annule, etc.

**:redo** refait une opération venant d'être annulée

`Ctrl-l` actualisation de l'écran

**.** répétition de la dernière modification

**:!cmd** soumission d'une commande *cmd* au shell sans sortir de **vi**

**!!cmd** exécute *cmd* et place le résultat de la commande sur la ligne courante.

### A.5.9.a Aide de l'éditeur

**:help** [*mot*] affiche l'aide sur la rubrique *mot*

### A.5.9.b Coloration syntaxique

**:sy[n[tax]]** (**enable** | **on** | **off**) si **enable** ou **on**, active la coloration syntaxique, et **off** la désactive. Par défaut, la coloration syntaxique est activée dans les versions récentes de **vi**.

**vi** tente alors de deviner le type de contenu du fichier afin d'utiliser la bonne coloration syntaxique. Il fixe alors une valeur aux options **filetype** et **syntax**. Cette dernière peut être modifiée comme suit si elle ne correspond pas :

**:set syntax=(c|cpp|csh|java|sh|perl|php|lex|yacc|...)**

Il y a beaucoup d'autres choix possibles, pratiquement un par fichier d'extension `.vim` contenu dans le répertoire `/usr/share/vim/syntax`.

### A.5.9.c Options de l'éditeur

Ces options doivent être demandées à partir du mode commandes de **vi**.

<b>:set all</b>	affichage des options disponibles
<b>:set</b>	affichage des options qui sont activées
<b>:set tabstop=<i>n</i></b>	fixe à <i>n</i> le nombre d'espaces qui forme une tabulation
<b>:set expandtab</b>	les tabulations qui seront insérées dans le futur seront remplacées par les espaces qu'elles représentent (et indiqué par l'option <b>tabstop</b> ). Il sera toujours possible d'insérer une réelle tabulation en tapant <span style="border: 1px solid black; padding: 0 2px;">Ctrl-v</span> <span style="border: 1px solid black; padding: 0 2px;">Tab</span>
<b>:set noexpandtab</b>	annule l'option <b>:set expandtab</b>
<b>:set paste</b>	lorsque que du texte est copié-collé, il sera inséré sans l'indenter
<b>:set ff=(dos unix)</b>	change le format d'enregistrement de fichier. <b>dos</b> correspond au format utilisé par Windows. La différence se situe notamment dans le codage des fins de lignes.
<b>:set number</b>	affichage des numéros de ligne du fichier (aussi <b>:set nu</b> )
<b>:set nonumber</b>	suppression de l'affichage des numéros de ligne du fichier (aussi <b>:set nonu</b> )
<b>:set autoindent</b>	active l'indentation automatique (aussi <b>:set ai</b> )
<b>:set noai</b>	désactive l'indentation automatique (aussi <b>:set noautoindent</b> )
<b>:set cindent</b>	active l'indentation automatique basée sur le langage C/C++
<b>:set nocindent</b>	désactive l'indentation automatique basée sur le langage C/C++
<b>:set shiftwidth=<i>n</i></b>	fixe à <i>n</i> le nombre d'espaces utilisés pour l'indentation demandée par l'option <b>cindent</b> ou les commandes <b>&gt;&gt;</b> , <b>&lt;&lt;</b> , <b>:&lt;</b> et <b>:&gt;</b> . La valeur par défaut est 8
<b>:set hls</b>	active la mise en évidence des chaînes recherchées trouvées
<b>:set nohls</b>	désactive la mise en évidence des chaînes recherchées trouvées
<b>:set ignorecase</b>	ne tient pas compte des minuscules et des majuscules lors d'une recherche
<b>:set list</b>	affichage des caractères de tabulation et de fin de ligne
<b>:set nolist</b>	désactive l'option <b>:set list</b>
<b>:set wrapscan</b>	permet la recherche cyclique sans tenir compte du début ou de la fin du fichier
<b>:set nowrapscan</b>	désactive la recherche cyclique

### A.5.9.d Fichier de configuration

Lorsque vi est exécuté, il recherche dans le répertoire d'accueil l'utilisateur si celui-ci contient le fichier **.vimrc** qui est un fichier de configuration. Ce fichier peut contenir différentes commandes que vi exécute au démarrage. Parmi ces commandes, il peut y avoir des commandes **set** (mais sans le **:**) comme celles ci-dessus.

## A.5.10 Facilités pour la programmation

Une première facilité est celle indiquée dans les déplacements pour rechercher les correspondances entre les (, {, [ et les ), }, ]. Les autres concernent la complétion automatique, la recherche de fonction/identificateur, l'indentation, etc.

### A.5.10.a Complétion automatique

`Ctrl-P` après avoir tapé le début d'un mot clé ou d'un identificateur, propose un mot clé/identificateur qui correspond. Taper à nouveau sur `Ctrl-P` pour la proposition suivante.

### A.5.10.b Recherche d'un identificateur

`[I` affiche les lignes où apparaît le mot sous le curseur. Chaque ligne affichée est précédée d'un numéro (voir ci-dessous).

### A.5.10.c Déplacement jusqu'à un identificateur et retour à la position initiale

`[n] [Tab` se place sur la ligne numéro  $n$  selon la liste affichée par `[I`

`Ctrl-O` après s'être déplacé avec la commande précédente, permet de revenir à la ligne initiale.

### A.5.10.d Indentation

`[n] >>` décale (indente) vers la droite les  $n$  lignes à partir du curseur. La valeur de l'option **shiftwidth** (par défaut 8) fixe le nombre d'espaces (ou de tabulations si l'option **expandtab** est désactivée) requis pour le décalage

`[n] <<` décale vers la gauche les  $n$  lignes à partir du curseur

`: [étendue] > { > }` décale les lignes spécifiées par l'*étendue* d'autant de niveaux vers la droite qu'il y a de `>`

`: [étendue] < { < }` décale les lignes spécifiées par l'*étendue* d'autant de niveaux vers la gauche qu'il y a de `<`

## A.6 Mode visuel

Le mode visuel n'existe que dans **vim**. Il sert à appliquer une commande à du texte sélectionné à l'aide des commandes de déplacement.



**La sélection de texte ne se fait pas avec la souris.**

Il y a 3 modes visuels dans lesquels on entre à partir du mode commandes en tapant :

**v** (mode visuel normal) où l'on pourra sélectionner du texte quelconque continu ;

**V** (mode visuel par ligne) où l'on ne pourra sélectionner que des lignes ;

`Ctrl-v` (mode visuel par bloc) où l'on pourra sélectionner des blocs ou régions (par exemple, un rectangle de texte sur plusieurs lignes).

Pour sortir du mode visuel sans taper de commande, il faut taper `ESC`. Sinon, taper l'une des commandes suivantes produit un effet et fait sortir de ce mode :

- y** pour copier le texte sélectionné
- d** pour supprimer le texte sélectionné
- >** pour décaler vers la droite de **shiftwidth** caractères les lignes sélectionnées
- <** pour décaler vers la gauche de **shiftwidth** caractères les lignes sélectionnées

**i** Bien d'autres possibilités existent. Taper **:help visual.txt** dans **vi** pour en savoir plus.



# Annexe B

## Gestion des patchs

---

### B.1 Introduction

L'utilisation des patch fait partie des exercices de style de plus en plus incontournables pour un informaticien. Il y a plusieurs utilisations des patch, notamment :

- corriger des applications ou un système, pour lesquelles des bogues ont été découverts après leur diffusion ;
- faire passer d'une version antérieure à une nouvelle version ;
- rajouter des fonctionnalités.

Un patch est un ensemble de modifications à faire subir à un ou plusieurs fichiers de façon à obtenir une nouvelle version de ces fichiers. Généralement, un patch est bien moins lourd que la nouvelle version elle-même (sans quoi, il est préférable de se procurer cette version) car les fichiers concernés changent assez peu d'une version à une autre. D'autre part, l'utilisation de patchs permet de garder une trace des modifications effectuées, et de revenir à une ancienne version si besoin.

Sous Linux, la création de patch se fait avec la commande **diff**, et leur application se fait avec la commande **patch**.

### B.2 diff : comparer des fichiers

La commande **diff** compare des fichiers et indique leurs différences. Elle est en particulier utilisée pour créer des fichiers patch qui seront ensuite exploités par l'utilitaire **patch**.

**diff** est très complète et on se limitera à une infime partie de ses possibilités.

#### Synopsis

```
diff [-curN] source cible
```

Normalement *source* et *cible* sont des fichiers ou des répertoires. Si *source* est un répertoire mais pas *cible*, le fichier *cible* est comparé au fichier correspondant de *source*. Si *source* et *cible* sont des répertoires, leurs fichiers de même nom sont comparés deux à deux.

L'option **-r** demande de faire cette comparaison récursivement. L'option **-N** demande à ce que si un fichier ne se trouve que dans un répertoire, considérer qu'il existe aussi dans l'autre, mais vide.

**diff** écrit le résultat des comparaisons sur la sortie standard. Il indique comment obtenir le deuxième fichier à partir du premier. La forme des différences constatées dépend de la présence d'options comme **-c** et **-u**. Sans ces options les différences sont écrites selon 3 possibilités :

- **na***deb*, *fin* : ajouter du texte. Dans le premier fichier, à partir de la ligne *n*, il faut ajouter les lignes *deb* à *fin* du deuxième fichier. Ces lignes sont ensuite écrites, précédées du signe **>**
- *deb*, **fin***na* : supprimer du texte. Du premier fichier, il faut supprimer les lignes de la ligne *deb* à la ligne *fin*. Elles sont en trop à partir de la ligne *n* du second fichier. Ces lignes sont ensuite écrites, précédées du signe **<**
- *deb*<sub>1</sub>, *fin*<sub>1</sub>**c***deb*<sub>2</sub>, *fin*<sub>2</sub> : remplacer du texte. Il faut remplacer les lignes *deb*<sub>1</sub> à *fin*<sub>1</sub> du premier fichier par les lignes *deb*<sub>2</sub> à *fin*<sub>2</sub> du second fichier. Les lignes du premier fichier sont ensuite écrites précédées par **<**, puis une ligne ne contenant que 3 tirets (**---**) est écrite, suivie des lignes du second fichier, précédées par **>**

L'option **-c** demande une sortie de **diff** en mode « contexte », où apparaissent notamment des informations sur les fichiers comparés. L'option **-u** demande une sortie en mode « unifié ». C'est généralement l'option utilisée pour créer des patches.

Avec l'option **-u**, pour chaque couple de fichiers comparés, **diff** écrit les deux lignes suivantes :

```
--- chemin/vers/fichier/ancien    informations-diverses
+++ chemin/vers/fichier/nouveau  informations-diverses
```

puis une série de contextes de modifications (appelées aussi *hunk*) à apporter au fichier "ancien".


Un *hunk* commence par une ligne de la forme :

```
@@ -debdép, ndép +debfin, nfin @@
```

indiquant que la modification doit porter sur *n<sub>dép</sub>* lignes à partir de la ligne numéro *deb<sub>dép</sub>*. Au final, on obtiendra un contexte qui commencera à la ligne *deb<sub>fin</sub>* et qui tiendra sur *n<sub>fin</sub>* lignes. Cette information permettra à **patch** d'effectuer la modification mais aussi de l'annuler.

Ensuite, la modification est écrite, indiquée par plusieurs lignes :

- celles commençant par un espace permettent de situer le contexte ;
- celles commençant par le caractère **-** sont à supprimer ;
- celles commençant par le caractère **+** sont à rajouter.

 Pour créer un fichier patch qui sera ensuite utilisé par **patch**, on spécifie généralement **-Nur** comme options et on redirige la sortie de **diff** dans le fichier patch.

## Exemples

```
$ cat ancien.cxx
```

```
/* Fichier : ancien.cxx */
/* version 1.0 */
/* Fichier ne contenant que du code C */
```

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

```

int main (int argc, char * argv [])
{
    int i = 10;

    while (i--)
    {
        printf (
            "Salut tout le monde\n"
        );
    }

} // main()

$ cat nouveau.cxx

/* Fichier : nouveau.cxx */
/* version 2.0 */
// Fichier adapté pour contenir du code C++

#include <iostream>
using namespace std;

int main (int argc, char * argv [])
{
    int i = 10;

    while (i--)
    {

        cout <<
            "Salut tout le monde\n"
            ;

    }

    return 0;

} // main()

```

### Sortie de diff en mode normal

```

$ diff ancien.cxx nouveau.cxx

1,5c1,3
< /* Fichier : ancien.cxx */
< /* version 1.0 */
< /* Fichier ne contenant que du code C */

```

```

<
< #include <stdio.h>
---
> /* Fichier : nouveau.cxx */
> /* version 2.0 */
> // Fichier adapté pour contenir du code C++
6a5,6
> #include <iostream>
> using namespace std;
14c14,15
<         printf (
---
>
>         cout <<
16c17
<         );
---
>         ;
19a21
>     return 0;

```

### *Sortie de diff en mode unifié*

```
$ diff -u ancien.cxx nouveau.cxx
```

```

--- ancien.cxx 2004-01-12 14:46:48.000000000 +0100
+++ nouveau.cxx 2004-01-12 14:46:28.000000000 +0100
@@ -1,9 +1,9 @@
-/* Fichier : ancien.cxx */
-/* version 1.0 */
-/* Fichier ne contenant que du code C */
-
-#include <stdio.h>
+/* Fichier : nouveau.cxx */
+/* version 2.0 */
+// Fichier adapté pour contenir du code C++

+#include <iostream>
+using namespace std;

    int main (int argc, char * argv [])
    {
@@ -11,12 +11,14 @@

        while (i--)
        {
-            printf (
+
+            cout <<

```

```

                "Salut tout le monde\n"
-                );
+                ;
    }

```

```

+    return 0;

```

```

} // main()

```

```

$ diff -u ancien.cxx nouveau.cxx > ancien-to-nouveau.patch

```

➡ création d'un fichier patch en redirigeant la sortie de **diff**

Sortie de diff en mode contexte

```

$ diff -c ancien.cxx nouveau.cxx

```

```

*** ancien.cxx 2004-01-12 14:46:48.000000000 +0100
--- nouveau.cxx 2004-01-12 14:46:28.000000000 +0100

```

```

*****

```

```

*** 1,9 ****

```

```

! /* Fichier : ancien.cxx */

```

```

! /* version 1.0 */

```

```

! /* Fichier ne contenant que du code C */

```

```

!

```

```

! #include <stdio.h>

```

```

    int main (int argc, char * argv [])
    {

```

```

--- 1,9 ----

```

```

! /* Fichier : nouveau.cxx */

```

```

! /* version 2.0 */

```

```

! // Fichier adapté pour contenir du code C++

```

```

+ #include <iostream>

```

```

+ using namespace std;

```

```

    int main (int argc, char * argv [])
    {

```

```

*****

```

```

*** 11,22 ****

```

```

        while (i--)
        {

```

```

!

```

```

        printf (

```

```

            "Salut tout le monde\n"

```

```

!

```

```

        );

```

```

    }

```

```

    } // main()
--- 11,24 ----

    while (i--)
    {
!
!        cout <<
!            "Salut tout le monde\n"
!            ;
    }

+    return 0;

    } // main()

```

□

### B.3 patch : appliquer ou annuler un patch

Sans entrer dans les détails, **patch** permet d'appliquer un patch créé avec la sortie de **diff** (voir exemples de la sortie unifiée de **diff**).

#### Synopsis

```
patch [-pn] [-bR] [fichier_origine [fichier_patch]]
```

```
patch [-pn] [-bR] < fichier_patch
```

La première forme est utile lorsqu'on ne veut appliquer le patch que sur le seul fichier *fichier\_origine* et que *fichier\_patch* contient les modifications à effectuer sur plusieurs fichiers.

La forme la plus couramment utilisée est la deuxième. *fichier\_patch* est un fichier contenant les modifications à apporter à un ou plusieurs fichiers (en général, toute une arborescence). Cela correspond à la sortie de **diff**, que ce soit avec l'option **-c**, l'option **-u** ou sous une autre forme (bien que l'option **-u** est normalement la plus utilisée pour cela).

**patch** transforme les fichiers indiqués dans *fichier\_patch* avec les modifications décrites dans *fichier\_patch*. Normalement, les fichiers ainsi modifiés sont remplacés, à moins d'utiliser l'option **-b** demandant d'effectuer une sauvegarde des fichiers d'origine. L'option **-R** demande de faire l'opération inverse : restaurer un fichier patché pour retrouver le fichier d'origine.

Lorsqu'un patch a déjà été appliqué sur un fichier et que **patch** est de nouveau invoqué avec le même patch, alors celui-ci demande si on ne veut pas en fait restaurer le fichier non patché. Si c'est le cas, **patch** restaure le fichier non patché et sauvegarde le fichier patché en lui ajoutant l'extension `.orig`.

L'option **-pn** sert lorsque l'on veut appliquer un patch sur des cibles qui ne sont pas tout à fait situées au même endroit que sur le système d'où vient le patch. Par exemple, si le *fichier\_patch* indique que le patch porte sur le fichier `chemin/vers/fichier` mais que ce fichier est situé en fait dans `vers/fichier`, il faut spécifier l'option **-p1**. Le chiffre *n* indique combien de répertoires il faut ignorer au début du chemin des fichiers

à patcher. Ceci parce que le créateur du patch dispose souvent d'arborescences de type `version-ancienne/arborescence-ancienne` et `version-nouvelle/arborescence-nouvelle`, ce qui n'est généralement pas le cas de la personne appliquant le patch. L'absence de l'option **-pn** demande d'ignorer la totalité du chemin et de ne garder que le nom des fichiers à patcher.

### Exemple

Pour cet exemple, on va avoir affaire à deux individus :

- bart qui est le développeur d'une application nommée **appli**. Il a déjà publié une version de son application, que l'utilisateur homer a installée, et vient de terminer une nouvelle version. bart étant organisé, la version précédente de son application est dans le répertoire `appli-v1`, alors que la nouvelle version est dans le répertoire `appli-v2`. Outre la publication (mise à disposition) de la version 2 de son application par l'intermédiaire d'une archive tgz, il décide de mettre aussi à disposition un patch permettant de mettre à jour la version 1, ce qui est moins lourd à télécharger.
- homer est un utilisateur qui a installé sur son système la version 1 de l'application de bart, et veut maintenant passer l'application en version 2 en utilisant le patch fourni par bart. On étudiera le cas où chez homer, l'application a été installée dans le répertoire `appli-v1` puis le cas plus probable où elle a été installée dans le répertoire `appli`.

#### Développeur bart

```
$ ls -lR appli-v1 appli-v2
appli-v1:
total 12
drwxr-xr-x    2 bart      simpson    4096 jan 12 15:33 docs
-rw-r--r--    1 bart      simpson         95 jan 12 15:34 README
drwxr-xr-x    2 bart      simpson    4096 jan 12 15:33 src

appli-v1/docs:
total 4
-rw-r--r--    1 bart      simpson     698 jan 12 15:21 cigale.txt

appli-v1/src:
total 4
-rw-r--r--    1 bart      simpson     265 jan 12 15:22 code.cxx

appli-v2:
total 12
drwxr-xr-x    2 bart      simpson    4096 jan 12 15:34 docs
-rw-r--r--    1 bart      simpson     182 jan 12 15:35 README
drwxr-xr-x    2 bart      simpson    4096 jan 12 15:35 src

appli-v2/docs:
total 4
-rw-r--r--    1 bart      simpson     777 jan 12 15:21 cigale.txt

appli-v2/src:
total 4
-rw-r--r--    1 bart      simpson     303 jan 12 15:22 code.cxx
$ diff -Nur appli-v1 appli-v2 > patch-v1-to-v2.patch
```

➡ Le fichier patch a été créé par bart et s'appelle `patch-v1-to-v2.patch`

Utilisateur homer qui dispose du patch et de l'application dans `appli-v1`

```
$ ls
appli-v1 patch-v1-to-v2.patch
$ patch < patch-v1-to-v2.patch
can't find file to patch at input line 4
Perhaps you should have used the -p or --strip option?
The text leading up to this was:
-----
|diff -Nur appli-v1/docs/cigale.txt appli-v2/docs/cigale.txt
|--- appli-v1/docs/cigale.txt      2004-01-12 15:21:12.000000000 +0100
|+++ appli-v2/docs/cigale.txt      2004-01-12 15:21:26.000000000 +0100
-----
File to patch: Ctrl-c
```

⇒ On arrête **patch** car les fichiers ne sont pas trouvés. En effet, si on n'utilise pas l'option **-p**, **patch** ignore la totalité des chemins. Il cherche donc à patcher le fichier `cigale.txt` qui n'est pas dans le répertoire de travail.

```
$ patch -p0 < patch-v1-to-v2.patch
patching file appli-v1/docs/cigale.txt
patching file appli-v1/README
patching file appli-v1/src/code.cxx
```

⇒ En spécifiant l'option **-p0**, la totalité du chemin est utilisée et **patch** va mettre à jour avec succès les fichiers `appli-v1/docs/cigale.txt`, etc.

✍ Contrairement à ce qu'on pourrait penser, l'application du patch n'a pas pour effet de renommer le répertoire `appli-v1` en `appli-v2` (ni de créer le répertoire `appli-v2`).

Utilisateur homer qui dispose du patch et de l'application dans appli

*C'est le cas le plus probable.*

```
$ ls
appli patch-v1-to-v2.patch
$ patch -p0 < patch-v1-to-v2.patch
can't find file to patch at input line 4
Perhaps you used the wrong -p or --strip option?
The text leading up to this was:
-----
|diff -Nur appli-v1/docs/cigale.txt appli-v2/docs/cigale.txt
|--- appli-v1/docs/cigale.txt      2004-01-12 15:21:12.000000000 +0100
|+++ appli-v2/docs/cigale.txt      2004-01-12 15:21:26.000000000 +0100
-----
File to patch: Ctrl-c
```

⇒ On arrête le patch car les fichiers ne sont pas trouvés. En effet, avec l'option **-p0**, **patch** veut patcher le fichier `appli-v1/docs/cigale.txt` or `appli-v1` n'existe pas. Il faut donc ignorer ce répertoire.

```
$ cd appli
```

⇒ il faut d'abord se placer dans le répertoire `appli`



```
$ patch -p1 < ../patch-v1-to-v2.patch
patching file docs/cigale.txt
patching file README
patching file src/code.cxx
```

⇒ puis patcher en utilisant l'option **-p1** pour ignorer le premier répertoire du chemin. Remarquons que le fichier patch se trouve à un niveau inférieur.

□



# Index

---

/dev/stderr, 265  
/dev/stdin, 265  
/dev/stdout, 265  
/etc/bash.bashrc, 110  
/etc/group, 136  
/etc/gshadow, 136  
/etc/passwd, 133  
/etc/profile, 108  
/etc/shadow, 133  
/etc/sudoers, 141  
/etc/updatedb.conf, 184  
/var/lib/mlocate/mlocatedb.conf, 184  
~/.bash\_profile, 109  
~/.bashrc, 110  
~/.bash\_logout, 111

addgroup, 137  
adduser, 134  
alias, 101  
apropos, 52  
atrm, 111  
awk, 163  
  
break, 254  
bunzip2, 187  
bzip2, 188  
bzip2recover, 188  
bzless, 188  
bzme, 188

case, 256  
cat, 64  
cd, 20  
chfn, 133  
chgrp, 41  
chmod, 36  
chown, 42  
chsh, 133

clear, 111  
cmp, 186  
compact, 184  
compress, 184  
continue, 255  
cp, 27  
cron, 184  
cupsd, 114  
curl, 176  
cut, 80

declare, 223  
df, 31  
diff, 291  
du, 31

echo, 62  
elif, 247  
else, 247  
emacs, 281  
enable, 44  
esac, 256  
exit, 239  
export, 273

false, 247  
fdisk, 18  
fi, 247  
file, 19, 173  
find, 178  
finger, 133, 144  
for, 249

getopt, 268  
getopts, 268  
gnome-terminal, 111  
gpasswd, 141  
grep, 153  
groupadd, 136  
groupdel, 137

groupmod, 137  
groups, 139  
gunzip, 186  
gzexe, 186  
gzip, 185  
  
head, 77  
help, 49  
  
id, 137  
if, 247  
info, 52  
  
jobs, 116  
  
kill, 128  
killall, 131  
  
last, 148  
less, 66  
let, 208  
ln, 191  
local, 235  
locate, 182  
lpadmin, 92  
lptions, 92  
lpq, 93  
lpr, 91  
lprm, 93  
lpstat, 92  
ls, 22  
lsof, 120  
  
mail, 95  
makewhatis, 52  
man, 50  
mesg, 200  
mkdir, 25  
more, 66  
mount, 17  
mv, 29  
  
nail, 100  
newgrp, 141  
nl, 74  
nohup, 129  
  
pack, 184  
passwd, 134  
paste, 74  
patch, 296  
pkill, 131  
ps, 116  
  
pstree, 120  
pwd, 21  
  
quota, 32  
  
read, 260  
return, 238  
rm, 30  
rmdir, 26  
rxvt, 111  
  
sed, 157  
select, 257  
shift, 266  
shuf, 74  
smbpasswd, 135  
sort, 83  
source, 228  
split, 174  
stat, 192  
su, 139  
sudo, 141  
  
tac, 74  
tail, 78  
tar, 188  
tee, 74  
test, 243  
top, 120  
tr, 75  
true, 247  
tty, 71  
type, 46  
  
umask, 39  
unalias, 103  
uncompress, 185  
uniq, 87  
unset, 210, 219  
until, 254  
unzip, 190  
updatedb, 184  
useradd, 134  
userdel, 134  
usermod, 134  
  
vi, 281  
vim, 281  
visudo, 141  
  
w, 145  
wc, 75  
wget, 178

whatis, 52  
whereis, 48  
which, 47  
while, 253  
who, 142  
whoami, 138, 198  
write, 200

zcat, 186  
zcmp, 186  
zdiff, 186  
zgrep, 186  
zip, 190  
zless, 186  
zmore, 186  
znew, 186