

UPMC

Master P&A/SDUEE

UE MNI (4P009)

Méthodes Numériques et Informatiques

Introduction à l'environnement Unix

Jacques.Lefrere@upmc.fr

Sofian Teber

Albert Hertzog

2017–2018

Table des matières

1	Introduction au système UNIX	13
1.1	Système d'exploitation	13
1.2	Historique d'unix	14
1.3	Principales caractéristiques du système UNIX	15
1.4	Compte utilisateur	16
1.5	Sessions unix	17
1.6	Exemples de commandes élémentaires d'affichage	17
2	Le shell : introduction	19
2.1	Syntaxe de la ligne de commandes	19
2.2	Aides à l'interactivité du shell	20

2.3	Générateurs de noms de fichiers	21
2.4	Documentation en ligne	23
3	Hierarchie des fichiers unix	24
3.1	Arborescence	24
3.2	Chemins d'accès (<i>path</i>) d'un fichier	27
3.2.1	Affichage du répertoire courant avec <code>pwd</code>	28
3.2.2	Changement de répertoire courant avec <code>cd</code>	28
3.3	Raccourcis pour les répertoires d'accueil	33
4	Commandes de base	34
4.1	Commandes de gestion de fichiers	34
4.1.1	Affichage de liste de noms de fichiers avec <code>ls</code>	34
4.1.2	Copie de fichiers avec <code>cp</code>	36

4.1.3	Déplacement et renommage de fichiers avec <code>mv</code>	37
4.1.4	Suppression de fichiers avec <code>rm</code>	38
4.1.5	Compression de fichiers avec <code>gzip</code> ou <code>bzip2</code>	39
4.2	Commandes de gestion de répertoires	40
4.2.1	Création de répertoire avec <code>mkdir</code>	40
4.2.2	Suppression de répertoire (vide) avec <code>rmdir</code>	40
5	Commandes pour fichiers textes	41
5.1	Fichiers binaires et fichiers texte, codage	41
5.2	Codage des fichiers textes	41
5.2.1	Transcodage de fichiers textes avec <code>recode</code> ou <code>iconv</code> . . .	42
5.3	Accès au contenu des fichiers	44
5.3.1	Identification des fichiers avec <code>file</code>	44

5.3.2	Comptage des mots d'un fichier texte avec <code>wc</code>	45
5.3.3	Affichage du contenu de fichiers texte avec <code>cat</code>	46
5.3.4	Affichage paginé du contenu d'un fichier texte avec <code>more/less</code>	47
5.3.5	Début et fin d'un fichier texte avec <code>head</code> et <code>tail</code>	48
5.3.6	Repliement des lignes d'un fichier texte avec <code>fold</code>	48
5.3.7	Affichage des différences entre deux fichiers texte avec <code>diff</code> .	49
5.3.8	Affichage de la partie texte d'un fichier binaire avec <code>strings</code>	49
5.3.9	Affichage d'un fichier binaire avec <code>od</code>	49
6	Environnement réseau	50
6.1	Connexion à distance via <code>slogin</code>	50
6.2	Transfert de fichiers à distance : <code>scp</code> , <code>sftp</code>	51
6.3	Explorateurs et téléchargement	52

7	Commandes avancées de gestion des fichiers	53
7.1	Recherche de fichiers dans une arborescence avec <code>find</code>	53
7.2	Archivage d'arborescence avec <code>tar</code>	59
7.3	Copies et synchronisation de fichiers avec <code>rsync</code>	64
8	Droits d'accès aux fichiers	66
8.1	Affichage des droits d'accès avec <code>ls -l</code>	67
8.2	Changement des droits d'accès avec <code>chmod</code>	68
9	Édition de fichiers textes	69
9.1	Les éditeurs sous unix et leurs modes	69
9.1.1	Éditeurs sous unix	69
9.1.2	Les modes des éditeurs	70
9.2	Principes de l'éditeur <code>vi</code>	71

10 Redirections et tubes	73
10.1 Flux standard	73
10.2 Redirections	74
10.2.1 Redirection de sortie vers un fichier (> et >>)	75
10.2.2 Redirection de l'entrée depuis un fichier (<)	77
10.3 Tubes ou <i>pipes</i> ()	78
10.4 Compléments	81
10.4.1 Redirection de la sortie d'erreurs vers un fichier (2> et 2>>)	81
10.4.2 Redirection de l'erreur standard vers la sortie standard (2>&1)	83
10.4.3 Les fichiers spéciaux : exemple /dev/null	84
10.4.4 Duplication de flux : tee	85
11 Filtres élémentaires	87

11.1 Définition	87
11.2 Classement avec <code>sort</code>	87
11.3 Transcription avec <code>tr</code>	90
11.4 Autres filtres élémentaires	91
12 Expressions régulières ou rationnelles	92
12.1 Signification des caractères spéciaux	92
12.2 Ancres	95
12.3 Ensembles de caractères	96
13 Le filtre <code>grep</code>	98
14 Le filtre <code>sed</code>	100
15 Le filtre <code>awk</code>	102

15.1	Structure des données pour <code>awk</code>	102
15.2	Structure d'un programme <code>awk</code>	103
15.3	Exemples de programmes <code>awk</code>	105
15.4	Mise en garde sur les caractères non-imprimables	106
16	Gestion des processus	109
16.1	Généralités : la commande <code>ps</code>	109
16.2	Caractères de contrôle et signaux	112
16.3	Commande <code>kill</code>	113
16.4	Processus en arrière plan	113
17	Code de retour	114
17.1	Code de retour	114
17.2	Combinaison de commandes <code>&&</code>	115

18 Variables shell	117
18.1 Affectation et référence	117
18.2 Extension de la portée d'une variable : variables d'environnement . . .	118
18.3 Variables de localisation (langue, ...)	121
19 Caractères interprétés par le shell	122
19.1 Substitution de commande	122
19.2 Métacaractères du shell	124
20 Shell-scripts	127
20.1 Fichiers de commandes ou shell-scripts	127
20.2 Les paramètres des scripts	128
20.3 Utilisation de la commande <code>set</code>	133

21 Structures de contrôle en shell (sh)	134
21.1 Introduction	134
21.2 Conditions	135
21.2.1 Structure <code>if . . . fi</code>	135
21.2.2 Structures <code>if</code> imbriquées : <code>elif</code>	137
21.2.3 Énumération de motifs (cas) : <code>case ... esac</code>	139
21.3 Les structures itératives	142
21.3.1 La structure <code>for . . . do . . . done</code>	142
21.3.2 La structure <code>until . . . do . . . done</code> (<i>jusqu'à ce que</i>)	146
21.3.3 La structure <code>while . . . do . . . done</code> (<i>tant que</i>) . . .	148
21.4 Compléments : branchements	150
21.4.1 La commande <code>exit</code>	150
21.4.2 La commande <code>break</code>	150

21.4.3 La commande <code>continue</code>	153
21.4.4 Redirections et boucles	155
22 Exemple commenté d'un script	157
22.1 Introduction	157
22.2 Le cœur de script	157
22.3 Version minimale du script	159
22.4 Version élémentaire du script	161
22.5 Version plus robuste du script	164
22.6 Limitations	168
23 Compléments sur le shell	169
23.1 Commandes internes	169
23.2 Exécution dans le shell courant	169

23.3	Exécution avec les droits admin via <code>sudo</code>	169
23.4	Autres commandes internes	170
23.4.1	La commande <code>eval</code>	170
23.4.2	La commande <code>exec</code>	171
23.5	Divers	171
23.5.1	Alias du shell	171
23.5.2	Identifier une commande <code>type</code>	172
23.5.3	Affichage d'une progression arithmétique <code>seq</code>	172
23.5.4	Récursivité	173
23.5.5	Fichiers d'initialisation du shell	174
23.6	Automatisation des tâches avec la commande <code>make</code>	175

1 Introduction au système UNIX

1.1 Système d'exploitation

- ensemble de programmes d'un ordinateur servant d'**interface** entre le matériel et les logiciels applicatifs
- abrégé S.E. (en anglais *operating system* O.S.)
- exemples : MS-DOS, Windows (XP, 7, ...,10), famille **Unix** (**linux**, Mac-OS, ...)

Linux aujourd'hui dominant dans le calcul intensif :

plus de 97% des calculateurs du TOP 500

N.-B. : **machine virtuelle** = application qui émule un système d'exploitation

⇒ ex. : linux émulé sous `virtualbox` ou `VMware` dans une fenêtre windows

1.2 Historique d'unix

- depuis les années 1970, plusieurs branches de développement
 - ⇒ quelques différences dans les commandes surtout au niveau administration
 - système ouvert : implémentations du téléphone portable au super-calculateur
 - propriétaires (aix, hp-ux, solaris, os-X, ...)
 - libres (**linux** depuis 1991, net-bsd, free-bsd, ...) : linux est (presque) un unix !
- nombreuses distributions linux, principales branches :
- **debian** ↪ ubuntu ↪ mint
 - **slackware** ↪ Suse ↪ **OpenSuse** (LUTES)
 - **Red-Hat** ↪ Mandrake ↪ **mandriva** ↪ mageia,
 - ↪ **CentOS** (serveur appli1)
 - ↪ scientific-linux,
 - ↪ Fedora, ...

1.3 Principales caractéristiques du système UNIX

- multi-tâches (concurrentes et indépendantes)
- multi-utilisateurs (dont l'administrateur ou *root*)
 - ⇒ système d'**identification** et **droits** d'accès aux fichiers
- chaînage des processus par les **tubes** (pipes)
 - ⇒ assemblage d'outils élémentaires pour accomplir des tâches complexes
- Le **shell** est l'interface utilisateur du système d'exploitation.

bash : *Bourne Again SHell* (`sh` : shell historique de Bourne)

l'interpréteur de commandes (**shell**) intègre un **langage de programmation** avec variables, structures de contrôle...

⇒ programmes interprétés en shell = fichiers de commandes = *shell-scripts*

⇒ création de commandes par l'utilisateur

1.4 Compte utilisateur

- un **identifiant** (ou *login*) (unique)
- un **mot de passe** (ou *password*) confidentiel
- un **groupe** parmi ceux définis sur la machine
- un **répertoire d'accueil** personnel (ou *home directory*) où stocker ses fichiers
- un « **interpréteur de commandes** » (ou *shell*) : **bash**

Ces informations sont stockées dans un fichier système (souvent `/etc/passwd`)

Le mot de passe est crypté

⇒ l'administrateur ne peut pas retrouver un mot de passe oublié

Ressources limitées, par exemple par quota sur le disque

⇒ problème de connexion en mode graphique si quota atteint.

1.5 Sessions unix

— point commun : une session commence par

- **identification** (*login*)
- **authentification** (*password*)

la même invite apparait après la fin de session

— deux types de **sessions** de travail :

- mode **texte** (console, accès distant (`slogin`), ...) : ligne de commande
- mode **graphique** (multi-fenêtres) : icônes et menus pour lancer les applications
(dont les consoles **konsole** et `xterm` par exemple)
environnements de bureau : **kde**, `gnome`, **xfce**, **lxde**...
gestionnaires de fenêtres : `fvwm`, `icewm`...

1.6 Exemples de commandes élémentaires d’affichage

la commande	affiche
date	la date
whoami	le login
hostname	le nom de la machine
who	la liste des utilisateurs connectés
echo "chaîne de caracteres"	la chaîne saisie
id	le numéro d'utilisateur
uname	le nom du système d'exploitation

2 Le shell : introduction

Le shell est un programme qui interprète les commandes saisies dans un terminal.

2.1 Syntaxe de la ligne de commandes

Le shell découpe la ligne de commande en mots séparés par des blancs

- (1) premier mot = **la commande** l'action
- (2) mots suivants = **les paramètres** ou arguments les objets
(rôle déterminé par leur position dans la ligne de commande)
- (3) paramètres **optionnels** introduits par « - » les modalités

cp

(1) commande

copie

-p

(3) option

en gardant la date

fich1

(2) paramètre 1

source

fich2


(2) paramètre 2

cible

- Le shell distingue les **majuscules** (rares) des **minuscules**
- il interprète certains **caractères** dits **spéciaux** : blancs, " , ' , \ , * , ? , ...
avant de d'exécuter la commande

 ⇒ éviter les blancs dans les noms de fichiers

2.2 Aides à l'interactivité du shell

 et 

permettent de parcourir l'historique des commandes

 et 

déplacements pour éditer la ligne de commande

^E ou **^A**

déplacement en fin (**End**) ou début de ligne (**^B** est pris *Back*)

TAB

demande au système de compléter le nom de commande ou de fichier

⇒ évite les fautes de saisie et **valide les chemins**

TAB **TAB**

affiche les différentes possibilités de complétion

plus beaucoup d'autres (voir chapitres suivants)

2.3 Générateurs de noms de fichiers

Caractères *jokers* interprétés par le shell pour désigner des fichiers selon des **motifs génériques**

- * une chaîne de caractères quelconque dans le nom d'un fichier
(y compris la chaîne vide)
- ? un caractère quelconque et un seul dans un nom de fichier
- [...] un caractère quelconque pris dans la liste exhaustive entre crochets
- [$c_1 - c_2$] un caractère quelconque entre c_1 et c_2 dans l'ordre lexicographique
- [!...] un caractère quelconque pris hors de la liste

Exemples de motifs de noms de fichiers

- *** tous les fichiers du répertoire courant (sauf ceux commençant par **.**)
- *.f90** tous les fichiers dont le nom finit par **.f90**
- *.*** tous les fichiers dont le nom comporte un point (au moins)
- data??** tous les fichiers dont le nom est **data** suivi de deux caractères
- f.[abc]** les fichiers **f.a**, **f.b**, et **f.c** s'ils existent
- f.[0-9]** les fichiers dont le nom s'écrit **f.** suivi d'un chiffre
NB. : **f.[25-70]** (maladroit, mais) les fichiers **f.0**, **f.2**, **f.5**, **f.6** et **f.7**
- f.[!0-9]** les fichiers dont le nom s'écrit **f.** suivi d'un caractère qui n'est **pas** un chiffre
- *.[ch]** les fichiers source en C (***.c**) et les fichiers d'entête (*header* : ***.h**)

Tester ces motifs avec par exemple la commande **echo** : `echo *. [ch]`


2.4 Documentation en ligne

— **man cmd** : affiche du manuel de la commande `cmd`

page par page grâce au filtre `more` ou **less**

– se déplacer dans le manuel :  , page suivante/précédente

– rechercher un motif : **/motif**

– sortir du manuel : touche  *quit*

Préciser parfois la section du manuel (1 = commandes, 3 = bibliothèques)

man 3 printf (⇒ la fonction C)

au lieu de `man printf` (⇒ section 1 commande)

— **cmd --help** : affiche un bref aide-mémoire de la commande

Rechercher quelle commande utiliser pour une opération : **man -k motclef**

3 Hiérarchie des fichiers unix

3.1 Arborescence

L'ensemble des fichiers est structuré hiérarchiquement en un **arbre unique** constitué de

- nœuds : **répertoires** (*directories*, dossiers (*folders*) sous windows),
 - feuilles : **fichiers** (*files*) ordinaires en général.
- ★ le séparateur de niveaux est la barre oblique / (*slash*)
- ★ le répertoire / est la **racine** (*root*), qui contient tous les autres fichiers.

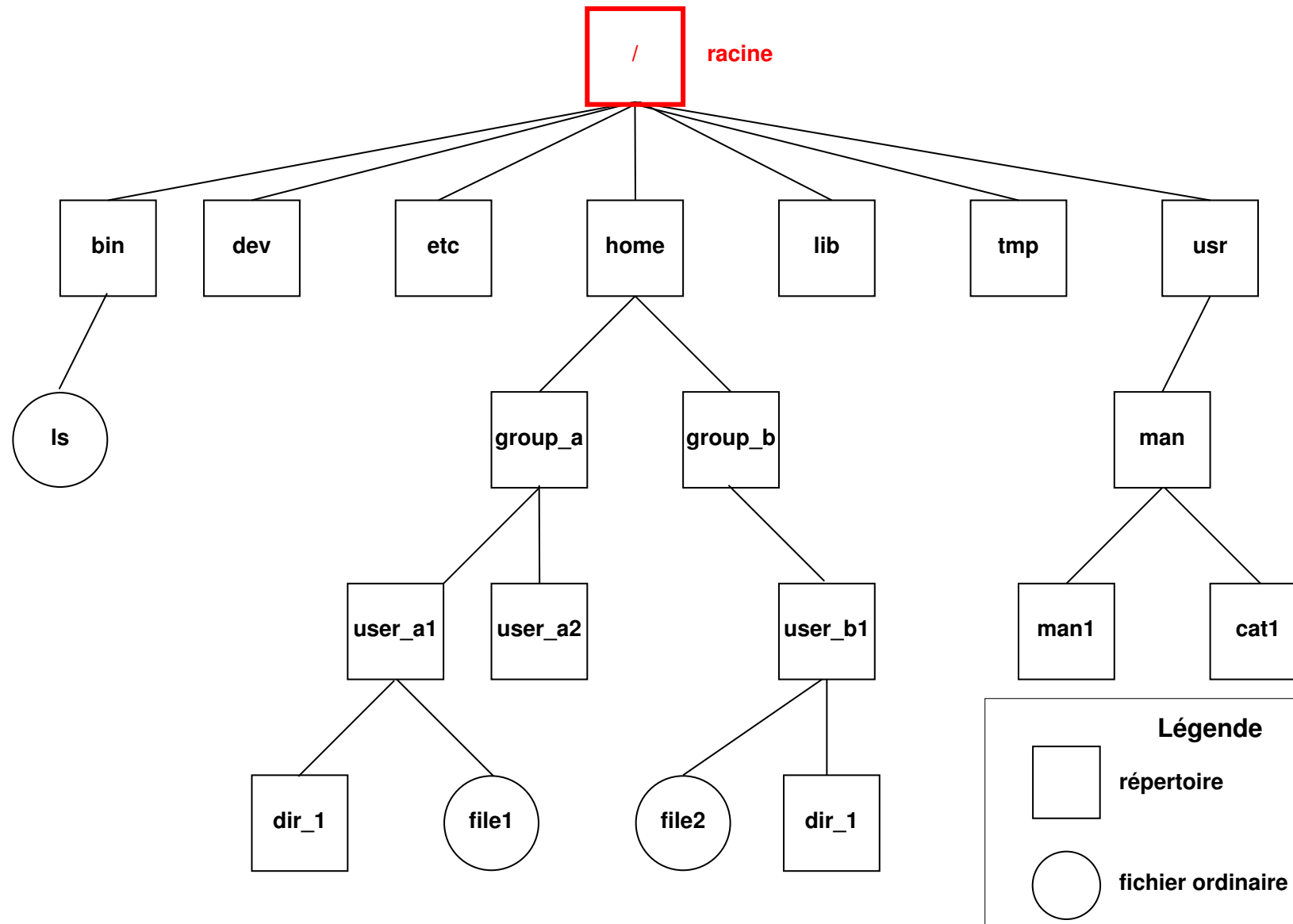


FIGURE 1 – Arborescence des fichiers UNIX

Montage de branches

L'arbre unique d'unix est **purement logique**.

Plusieurs périphériques peuvent être « montés » sur le système de fichiers
→ greffe temporaire d'une branche sur un «point de montage» de cet arbre.

Par exemple, un CDROM, un disque amovible ou une clef USB peuvent être « montés » dans les répertoires `/media/cdrom` ou `/media/removable`.
Après utilisation, il faut «démonter» ces branches (pour achever les transferts de données) avant de déconnecter physiquement ces périphériques.

Partage via le réseau : divers protocoles (`ssh` via **sshfs**, `nfs`) réalisent des montages pour partager des répertoires hébergés par d'autres ordinateurs (serveurs) via le réseau.

Sous windows séparateur = contre-oblique `\` (*antislash*)

périphériques désignés par une lettre suivie de « : » ex. `C : \` ou `D : \`



pas de distinction minuscule/majuscule ⇒ problème si montage linux

3.2 Chemins d'accès (*path*) d'un fichier

- **le chemin absolu** : commence toujours par `/` et comporte la liste complète des répertoires traversés **depuis la racine**,

Exemples : `/usr/man/man1/lis.1`, `/home/group_a/user_a1`

- **un chemin relatif** : comporte la liste des répertoires à parcourir **depuis le répertoire courant** jusqu'au fichier ou répertoire choisi.

Il ne commence jamais par `/` et doit passer par un nœud commun à la branche de départ (répertoire courant) et la branche d'arrivée.

- . **répertoire courant** ou de travail (*working directory*)
- .. **répertoire père** (*parent directory*)

 Des fichiers de même nom ne peuvent exister que dans des répertoires différents

NB : **tree** `rep` permet de représenter la branche qui part du répertoire `rep`

3.2.1 Affichage du répertoire courant avec `pwd`

pwd (*print working directory*) affiche le chemin **absolu** du répertoire courant
commande interne (*builtin*) du shell

3.2.2 Changement de répertoire courant avec `cd`

cd [*répertoire*] (*change directory*)

commande interne (*builtin*) du shell

cd (sans paramètre) retour au répertoire d'accueil `~/` .

cd `-` retour au précédent répertoire (dans le temps)

cd `..` retour au répertoire père (dans la hiérarchie)

Exemples en supposant que `pwd` affiche `/home/group_a/user_a1`

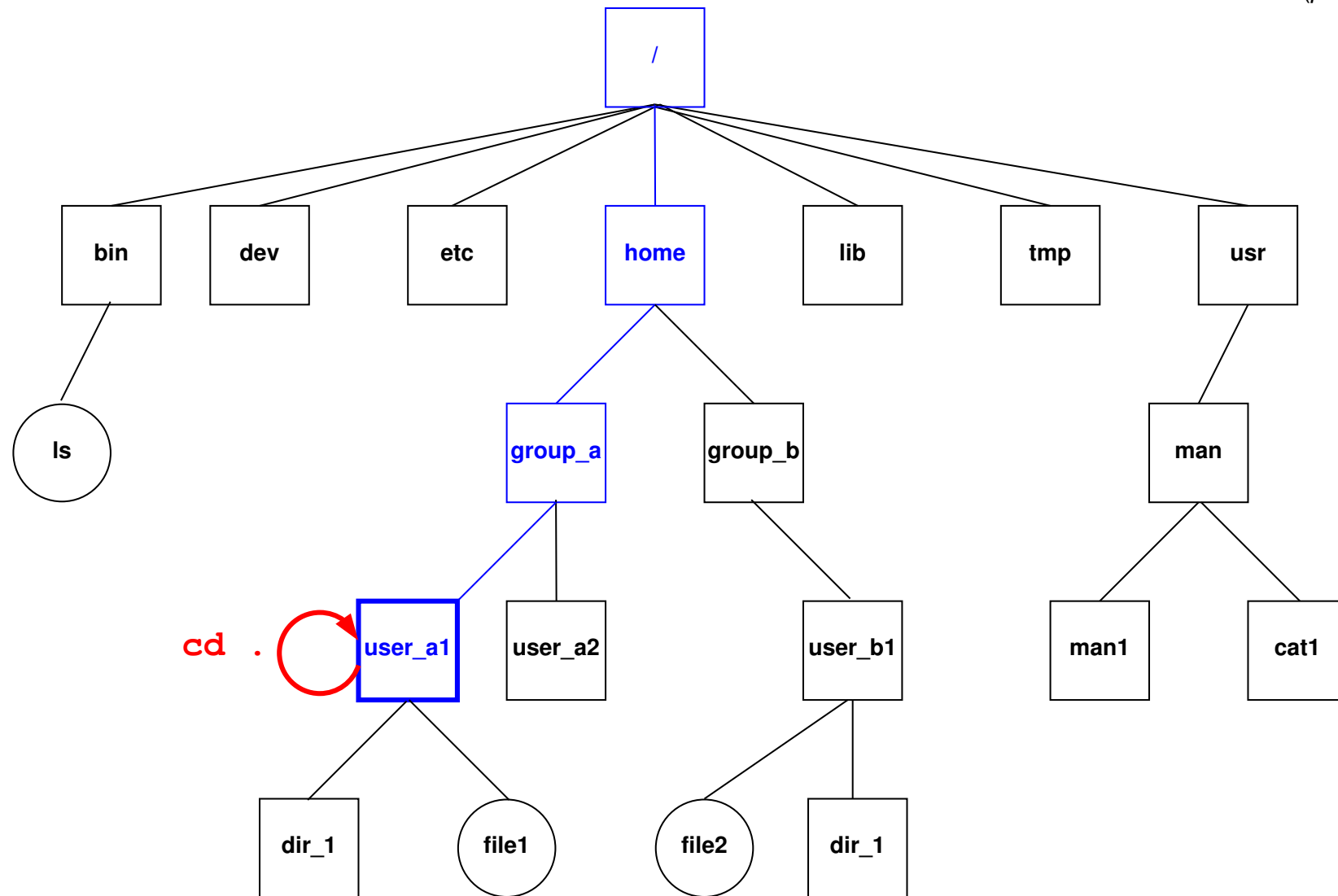


FIGURE 2 – La commande **cd .** laisse dans le répertoire courant `/home/group_a/user_a1`.

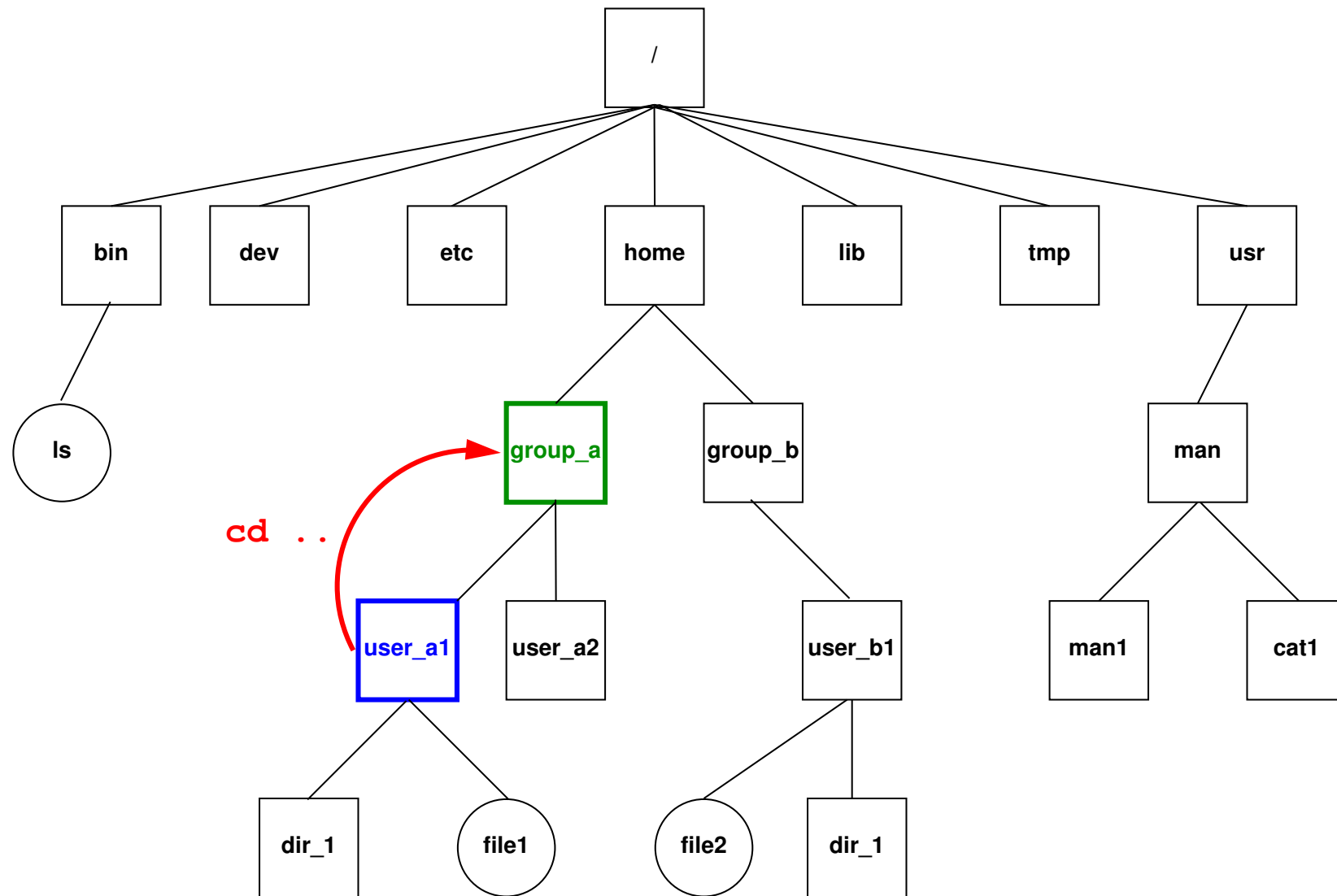


FIGURE 3 – La commande **cd ..** déplace dans le répertoire père `group_a`.

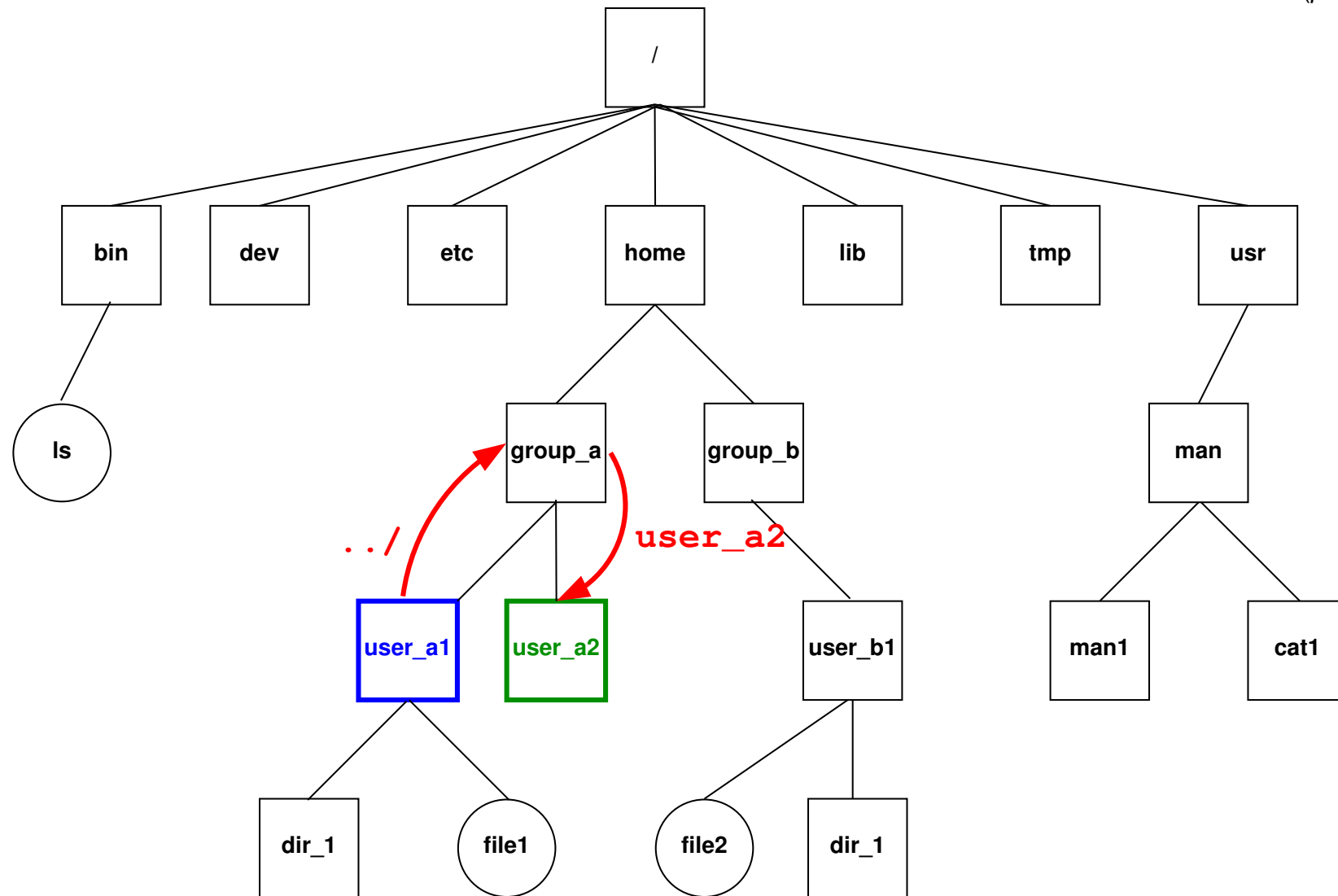


FIGURE 4 – La commande `cd ../user_a2` déplace dans le répertoire `user_a2`

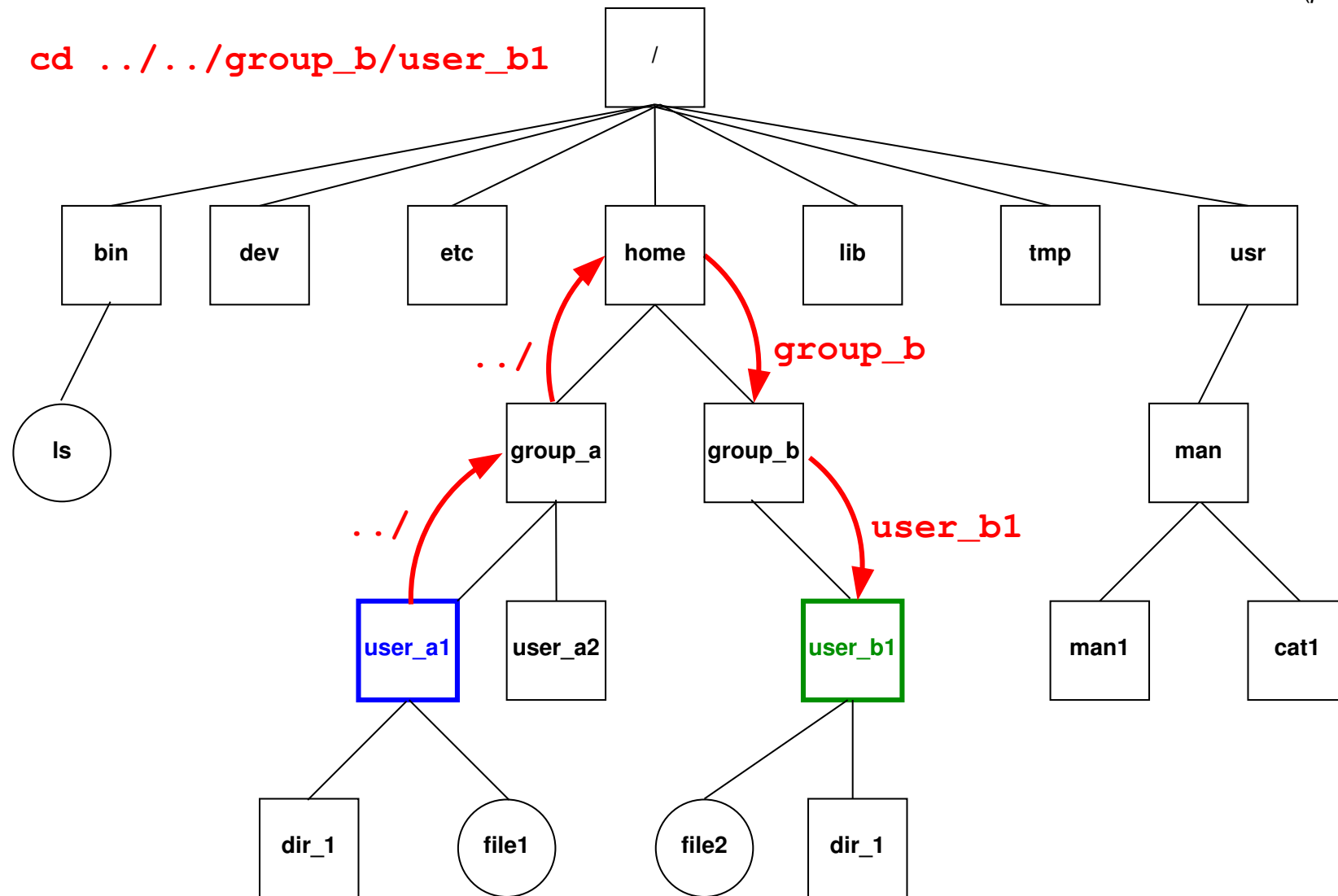


FIGURE 5 – La commande `cd ../../group_b/user_b1` déplace dans le répertoire `user_b1`.

3.3 Raccourcis pour les répertoires d'accueil

Chemins en fait absolus :

~*user* répertoire d'accueil de l'utilisateur nommé ***user***

~ son propre répertoire d'accueil

Exemples :

`~/ .bash_profile`

est le chemin absolu de votre fichier d'initialisation personnel.

`~lefrere/M1/Doc/unix/poly-unix/`

est le chemin absolu du répertoire du polycopié UNIX,
situé sous le compte de l'utilisateur `lefrere`.



Ne pas confondre `~lefrere` et `~/lefrere`

4 Commandes de base

 Les arguments fichier des commandes pourront comporter un chemin d'accès .

4.1 Commandes de gestion de fichiers

4.1.1 Affichage de liste de noms de fichiers avec `ls` (*list*)

```
ls [-options] [liste_de_fichiers]
```

Quelques options :

- a** (*all*) liste aussi les fichiers cachés (de nom commençant par `.`)
- l** (*long*) affiche les attributs (droits, taille, date, ...) des fichiers
- R** (*Recursive*) affiche la liste des fichiers contenus dans tous les sous répertoires éventuels
- d** (*directory*) affiche le nom des répertoires mais pas leur **contenu**

Exemples

`ls` (sans argument) liste (courte) des fichiers du répertoire courant \Leftrightarrow `ls .`

`ls -l` liste des fichiers du répertoire courant avec attributs

`ls -al` idem avec aussi les fichiers cachés (commençant par `.`)

 cas des répertoires : par défaut le **contenu**

`ls -l rep` liste détaillée des fichiers du répertoire `rep` (**contenu**)

`ls -dl rep` nom et attributs du **répertoire** `rep` (**contenant**)

`ls -l fic1 fic2 rep` liste détaillée des fichiers ordinaires `fic1`, `fic2`
et du **contenu du répertoire** `rep`

`ls -R /tmp` liste des fichiers de la branche `/tmp`

4.1.2 Copie de fichiers avec `cp` (*copy*)

— copie avec changement de nom éventuel (deux arguments seulement)

```
cp [-options] fichier_origine fichier_cible
```

— copie d'un ou plusieurs fichiers sans changement de nom

vers **un même répertoire**

```
cp [-options] liste_de_fichiers répertoire_cible
```

```
cp *.c bck/ copie les fichiers source C dans le répertoire bck
```

Principales options :

-r (*recursive*) copie d'une branche (si le premier argument est un répertoire)

-p (*permissions*) sans changer les droits ni la date

-i (*interactive*) demande de confirmation si la cible existe déjà

Confirmation en cas d'écrasement : répondre **y** (o si francisé)

4.1.3 Déplacement et renommage de fichiers avec mv (*move*)

Trois syntaxes possibles

1. **mv *fichier_origine fichier_cible***

2 arguments seulement renommage sauf si chemins d'accès différents

`mv fic1 fic2` : renomme fic1 en fic2

`mv /tmp/fic1 fic2` : idem mais prend fic1 dans /tmp

2. **mv *liste_de_fichiers repertoire_cible***

déplacement des fichiers de la liste vers le répertoire cible (qui doit exister)

`mv fic1 fic2 /tmp/` : déplace fic1 et fic2 dans /tmp

3. **mv *repertoire_source repertoire_cible***

renommage ou déplacement de branche

Principale option :

-i (*interactive*) demande de confirmation interactive si écrasement de fichier

4.1.4 Suppression de fichiers avec `rm`

en anglais *remove*

```
rm [-options] liste_de_fichiers
```

Principales options :

-i (*interactive*) demande de confirmation interactive

-r ou **-R** (*recursive*) destruction d'une branche (puissant mais... dangereux)

-f (*force*) sans demande de confirmation ni erreur si fichier inexistant



Attention : pas toujours de confirmation (sauf alias en **rm -i**)
destruction immédiate irréversible (pas de corbeille)

4.1.5 Compression de fichiers avec `gzip` ou `bzip2`

Compression et décompression sans perte d'information

— Compression → fichier de suffixe `.gz`

`gzip` *[-options]* *liste_de_fichiers*

— Décompression d'un fichier de suffixe `.gz`

`gunzip` *[-options]* *liste_de_fichiers*

Autre outil, plus efficace : `bzip2` / `bunzip2` (suffixe `.bz2`)

4.2 Commandes de gestion de répertoires

4.2.1 Création de répertoire avec `mkdir`

mkdir *répertoire* (*make directory*)

option `-p` (*parent*) : crée les répertoires parents si nécessaire

exemple : `mkdir -p dir/subdir`

4.2.2 Suppression de répertoire (vide) avec `rmdir`

rmdir *répertoire* (*remove directory*)

refus de suppression si le répertoire contient des fichiers

⇒ utiliser `rm -R répertoire`, mais dangereux !

5 Commandes traitant le contenu des fichiers texte

5.1 Fichiers binaires et fichiers texte, codage

Un **fichier** (ordinaire) = lot d'informations, conservé dans une mémoire permanente (disque, CD, clef USB, ...) et auquel on donne un nom.

Deux aspects du fichier :

bas niveau : suite de **bits** groupés en octets

haut niveau : représentation de texte, d'image, de code machine, ...
selon un certain **codage** qui permet d'interpréter la suite de bits.

Préférer nommer avec des suffixes rappelant le type de codage utilisé :

— fichiers **texte**

.**c** source C, **.f90** source fortran, **.txt** texte, **.html** hypertexte, ...

— fichiers **binaires**

.pdf pour du PDF, **.jpg** pour une image JPEG

.o pour un objet binaire, **.a** pour une bibliothèque, ...

5.2 Codage des fichiers textes

Plusieurs codages pour les caractères :


- **ASCII** sur 7 bits ($2^7 = 128$ caractères) => non accentués
- codages sur 1 octet = 8 bits ($2^8 = 256$ caractères) avec caractères accentués :
 - propriétaires : CP852, CP1252 sous windows, MacRoman sous MacOS
 - **ISO-8859** avec les variantes locales
ISO-8859-1 ou **latin1** pour le français par exemple
- évolution en cours vers standard **unicode** pour représenter toutes les langues :
nécessiterait jusqu'à 4 octets par caractère : UTF-32 !
implémentation **UTF-8** : taille variable des caractères : de 1 à 4 octets
 - sur-ensemble de l'ASCII (donc sur 1 octet pour les non-accentués)
 - les caractères non-ascii de latin1 sur 2 octets
- ⚠ – les codes binaires (sur 1 octet) des caractères accentués de latin1
sont **invalides** en UTF-8 !

5.2.1 Transcodage de fichiers textes avec `recode` ou `iconv`

— **iconv** **-f** *code_initial* **-t** *code_final* *fichier*

 la conversion s'arrête à la première combinaison invalide

— **recode** *code_initial..code_final* *fichier*

 par défaut `recode` travaille « en place » (modifie le fichier initial).

Exemples de transcodage de latin 1 vers utf-8 :

```
iconv -f ISO-8859-1 -t UTF-8 < fic-iso.txt > fic-utf8.txt
```

```
recode 'ISO-8859-1..UTF-8' < fic-iso.txt > fic-utf8.txt
```

De nombreux éditeurs (`vim`, `emacs`...) peuvent faire de la conversion au vol pour la phase d'édition, puis sauvegarder dans le codage initial.

 Ne pas mélanger deux codages dans un fichier (via par ex. copier/coller)

5.3 Accès au contenu des fichiers

5.3.1 Identification des fichiers avec `file`

file *liste_de_fichiers*

affiche une indication sur la nature du fichier (texte, binaire, ...)

⇒ l'utiliser pour savoir avec quelles commandes manipuler un fichier

```
a.out: ELF 64-bit LSB executable, x86-64
carre.f90: ASCII text
carre+invite.c: symbolic link to `carre+invite-utf.c'
carre+invite-iso.c: ISO-8859 C program text
carre+invite-utf.c: UTF-8 Unicode C program text
ligne.txt: ISO-8859 text
ligne.utf: UTF-8 Unicode text
poly-unix.tex: LaTeX 2e document text
tel-arbre.pdf: PDF document, version 1.5
```

5.3.2 Comptage des mots d'un fichier texte avec `wc`

```
wc [-cmwl] [liste_de_fichiers] (wordscount)
```

Affiche par défaut le nombre de lignes, de mots et d'octets, sauf si options cumulables pour sélectionner :

-l compte les lignes (*lines*)

-w compte les mots (*words*)

-m compte les caractères (*multibytes*, **c** est pris !) : utile en UTF-8 seulement

-c compte les octets (*characters* au sens historique !) comme `ls -l`



ordre d'affichage **fixe l, w, m, c**, c'est-à-dire du plus gros ensemble au plus petit
affiche aussi les totaux si plusieurs fichiers

5.3.3 Affichage du contenu de fichiers texte avec `cat`

`cat` [*liste_de_fichiers*]


affiche (**concatène**) le contenu des fichiers de la liste

 pas de contrôle du défilement (voir `more` ou `less`)

ex : `cat fic1 fic2 fic3` concatène et affiche le contenu des trois fichiers

`cat` = filtre identité : recopie l'entrée standard (clavier) sur la sortie standard (écran)

`cat -n` affiche les lignes avec leur numéro en tête, suivi d'une tabulation

 Ne pas confondre `cat fichier` avec `echo chaine`

5.3.4 Affichage paginé du contenu d'un fichier texte avec `more/less`

more *liste_de_fichiers*

affiche le contenu des fichiers de la liste (avec contrôle du défilement)

less *liste_de_fichiers*

préférable sous linux (défilement arrière possible)

Requêtes sous le pagineur

Entrée	avance d'une ligne
--------	--------------------

Espace	avance d'un écran
--------	-------------------

/motif recherche la prochaine occurrence de *motif* en avançant

?motif recherche la prochaine occurrence de *motif* en reculant

q quitte l'affichage



(nécessaire avec **less** car on peut remonter)

Rappel : **less** = pagineur utilisé par la commande **man**

5.3.5 Début et fin d'un fichier texte avec `head` et `tail`

head/tail [*options*] [*liste_de_fichiers*]

head -n nb fichier affiche les nb premières lignes de fichier

tail -n nb fichier affiche les nb dernières lignes de fichier

tail -n +11 fichier affiche à partir de la ligne 11

5.3.6 Replieement des lignes d'un fichier texte avec `fold`

fold [*options*] [*liste_de_fichiers*]

-w width : longueur (80 caractères par défaut)

-s ne coupe pas les mots (replieement sur les espaces)

5.3.7 Affichage des différences entre deux fichiers texte avec `diff`

`diff` *fichier_1 fichier_2*

`vimdiff` pour éditer 2 fichiers en parallèle (changer de fenêtre : `^W w`)

5.3.8 Affichage de la partie texte d'un fichier binaire avec `strings`

`strings` [*options*] *fichier*

5.3.9 Affichage d'un fichier binaire avec `od`

`od` [*options*] [*liste_de_fichiers*]

octal dump

formats d'affichage introduits par `-t`

`-t d4` pour des entiers sur 4 octets `-t f4` pour des flottants sur 4 octets

6 Environnement réseau

6.1 Connexion à distance via `slogin`

Connexion sur une machine distante grâce à la commande sécurisée **slogin**.

Authentification sur la machine distante par mot de passe ou échange de clefs `ssh`.

slogin *user@dist_host.domain*

```
slogin etu1@sappli1.datacenter.dsi.upmc.fr
```



ne pas oublier le login, sauf si identique sur la machine locale

Option **-X** pour autoriser les applications graphiques (fenêtres X11) via `ssh`

Lancement de commandes sur la machine distante :

ssh *user@dist_host.domain dist_cmd*

```
ssh etu1@sappli1.datacenter.dsi.upmc.fr ls ~lefrere/M1/Doc
```

6.2 Transfert de fichiers à distance via `scp` et `sftp`

Copie de fichiers personnels entre deux machines, sans ouvrir de session sur la machine distante, via `scp` (fournir le mot de passe à chaque commande)

Syntaxe de `cp` mais préfixer le chemin d'accès des fichiers distants par

`user@dist_host.domain:`

`scp` `[user1@]host1:file1` `file2` distant vers local

`scp` `file1` `[user2@]host2:file2` local vers distant

Session `sftp` (*secure file tranfert protocol*) pour plusieurs transferts

`sftp user@dist_host.domain`

- Après authentification sur le serveur distant,
- navigation distante : `cd`
- navigation locale : `lcd`
- importation de fichiers distants : `get dist_file`,
- exportation de fichiers vers la machine distante : `put local_file`
- `exit` ou `quit` pour terminer la session `sftp`.

6.3 Explorateurs et téléchargement

Navigateurs Web (lynx, firefox, opera, konqueror, amaya, ...)

Protocoles : **ftp** (*File Transfer Protocol*), **http** (*Hypertext Transport Protocol*),
ou **https** (sécurisé par cryptage).

Ressources localisées grâce à une *URL (Universal Resource Locator)*.

Exemples d'*URL* :

file: /home/lefrere/M1/Doc/unix/ sur la machine locale

http: //www.formation.jussieu.fr/ars/2011-2012/UNIX/cours/

En ligne de commande : **wget** ou **curl** pour télécharger des fichiers

par exemple :

```
wget "http://ftp.g95.org/v0.92/g95-x86-linux.tgz"
```

7 Commandes avancées de gestion des fichiers

7.1 Recherche de fichiers dans une arborescence avec `find`

find *répertoire critère(s) action*

Recherche **récursive** dans toute la branche sous *répertoire*

Commande très puissante : **critères** de sélection nombreux

pouvant être combinés pour former une expression évaluée à vrai ou faux.

- name** *motif* nom selon un motif (à protéger du shell)
- iname** *motif* version de `-name` ignorant la casse
- size** *entier*[ckM] taille en octets (c), kilooctets (k), mégaooctets (M)
- newer** *fichier* plus récent qu'un fichier
- type** *T* de type donné (f=ordinaire, d=répertoire)

Actions les plus usitées :

-print affiche la liste des chemins des fichiers (un par ligne)

-ls affiche la liste des fichiers avec leurs attributs (comme `ls -dils`)

Exemples

```
find . -name a.out -print
```

affiche la liste des fichiers nommés `a.out` sous le répertoire courant
(dans toute la hiérarchie en dessous de ce répertoire)

```
find . -name "*.c" -print
```

(le shell ne doit pas interpréter le caractère `*`)

```
find /tmp -size +1000c -size -2000c -print
```

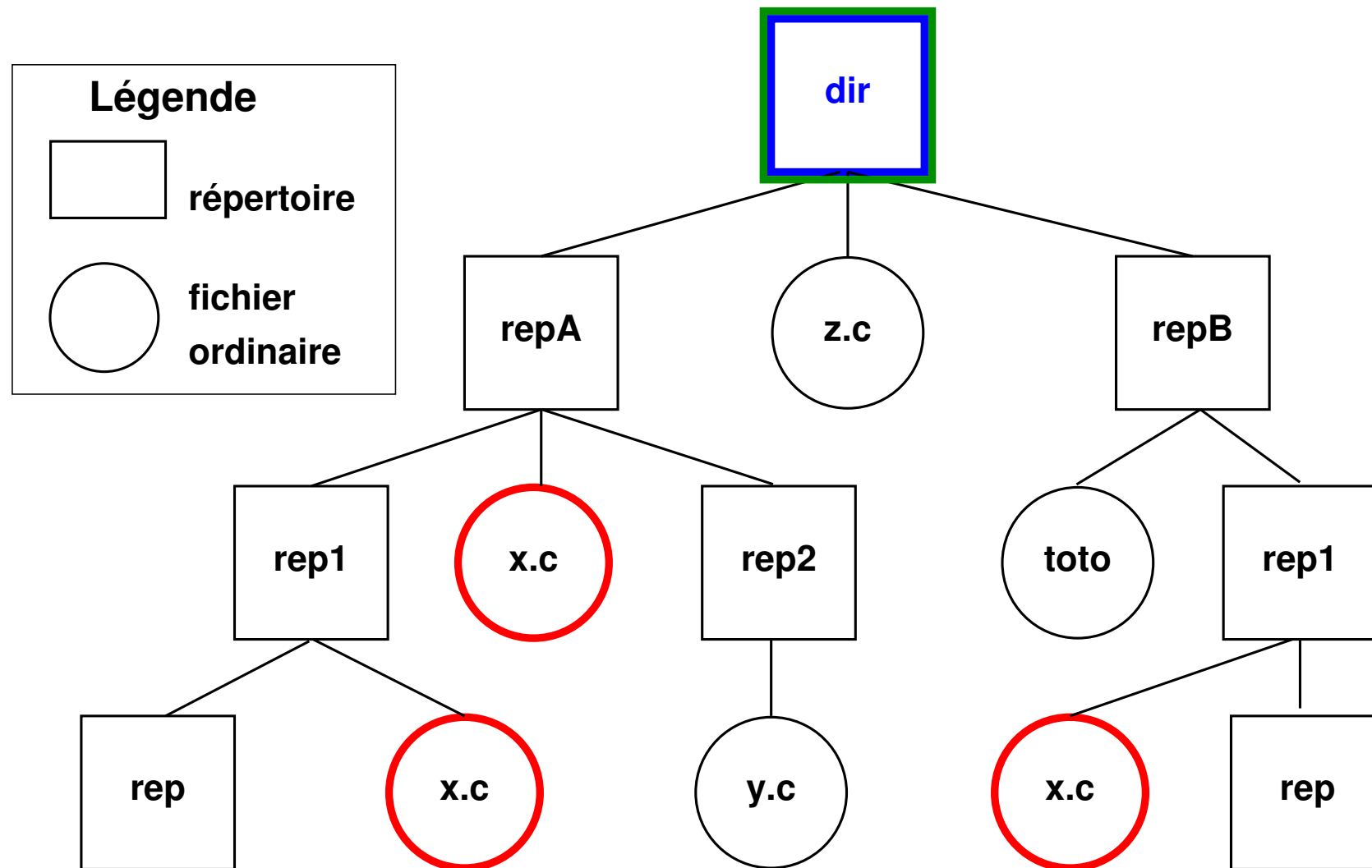
affiche la liste des fichiers de taille entre 1000 et 2000 octets sous `/tmp`



Ne pas oublier le premier argument de `find` : le noeud (répertoire) de départ

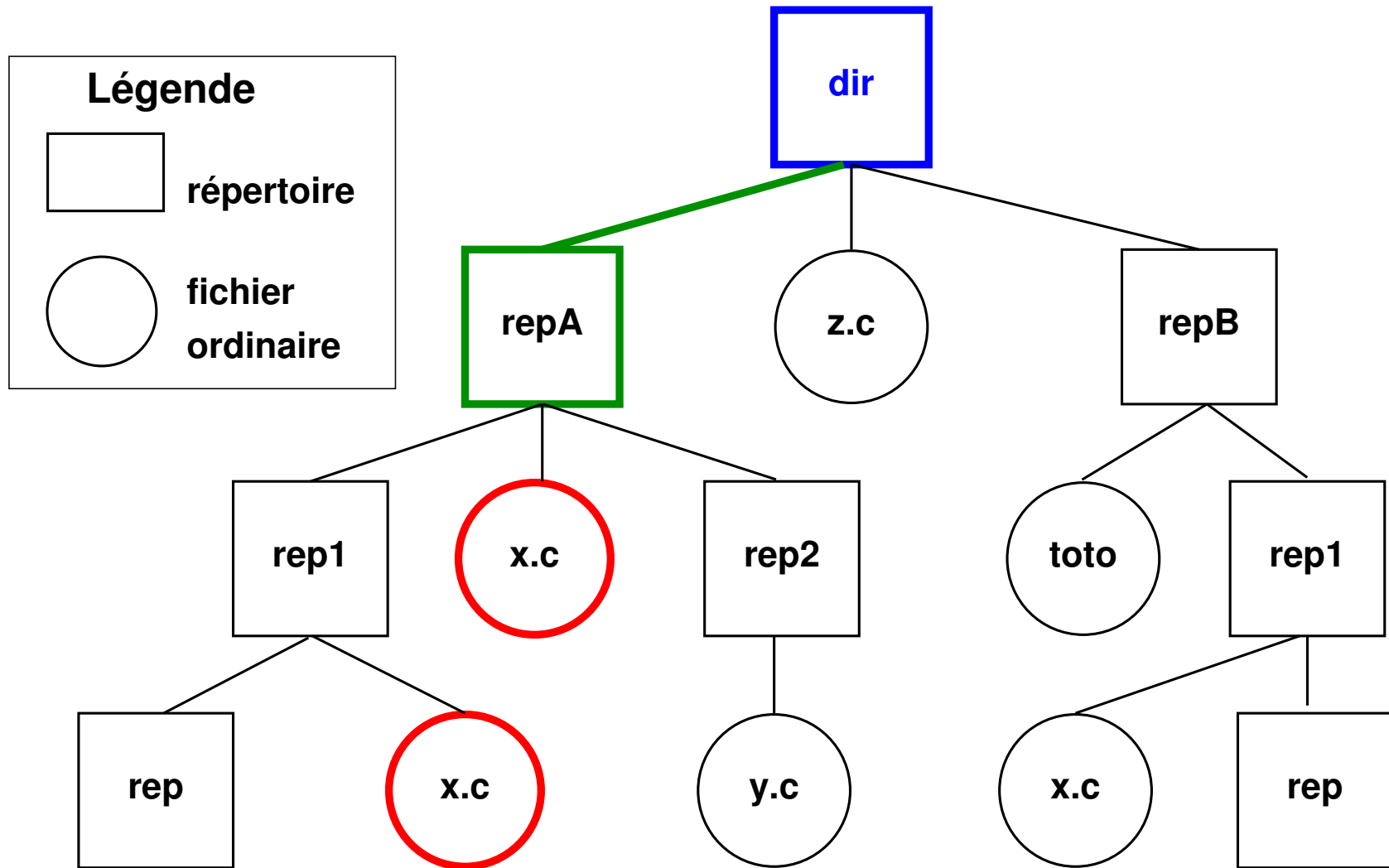


Ne pas confondre avec `ls -R`

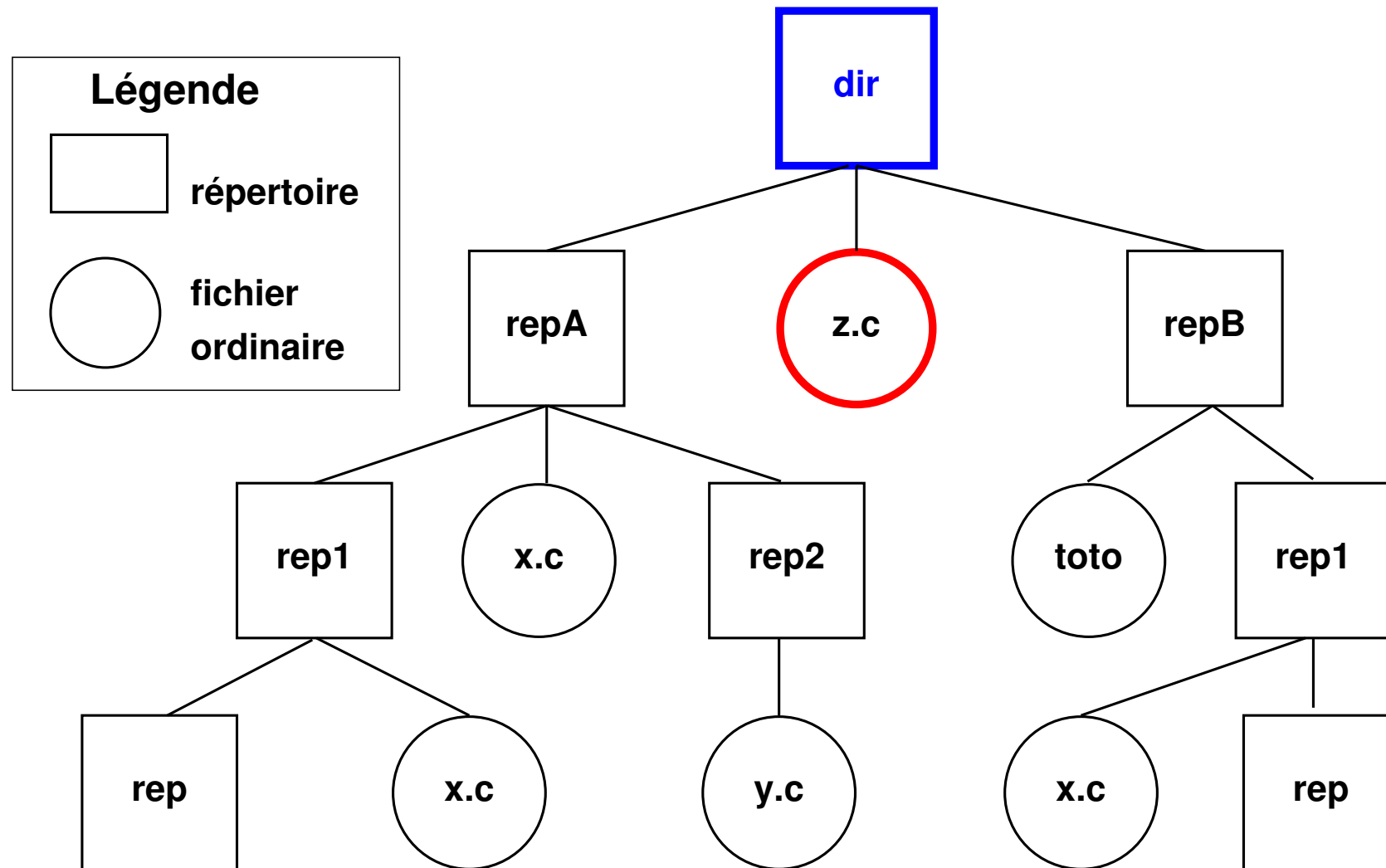


`find . -name x.c -print` si `dir` est le répertoire de travail

part de `.` donc de `dir` ⇒ trois fichiers



`find repA -name x.c -print` à partir de `repA` ⇒ deux fichiers



`find . -name *.c -print` (* interprété par le shell) \Rightarrow `z.c`

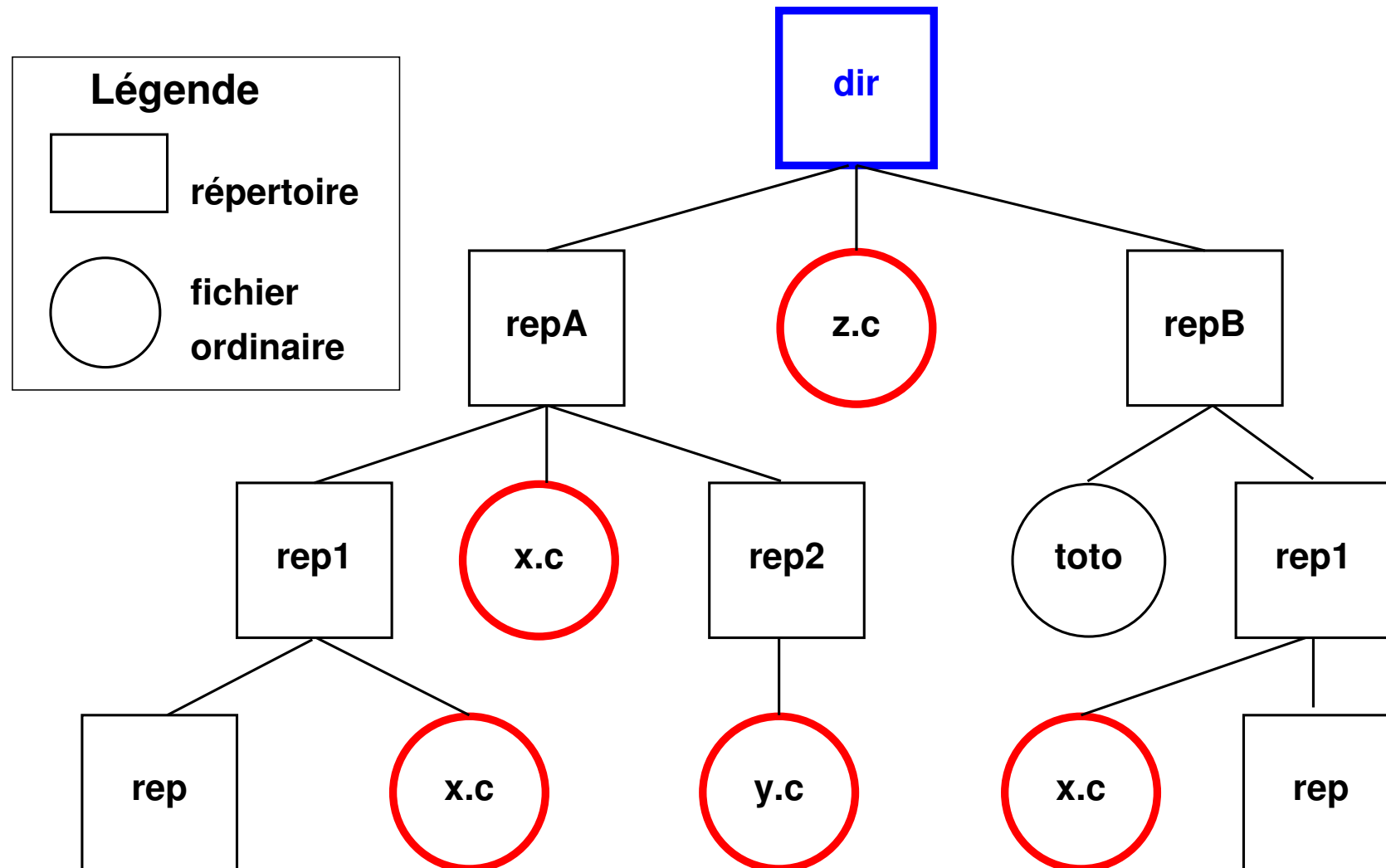


FIGURE 6 – `find . -name '*.c' -print`

⇒ cinq fichiers

7.2 Archivage d'arborescence avec `tar`

`tar` *options archive [répertoire]*

Principales **actions** possibles (une et une seule) :

- `-c`** (*create*) création de l'archive à partir de l'arborescence
- `-t`** (*list*) liste des fichiers tels qu'ils seront extraits
- `-x`** (*extract*) extraction des fichiers pour restaurer l'arborescence

Autres options combinables :

- `-v`** (*verbose*) affiche des informations complémentaires
- `-f`** *archive* (*file*) précise le nom du fichier d'archive utilisé (nécessaire)
- `z` ou `j`** avec dé/compression (`gzip` ou `bzip2`) du fichier `.tar`

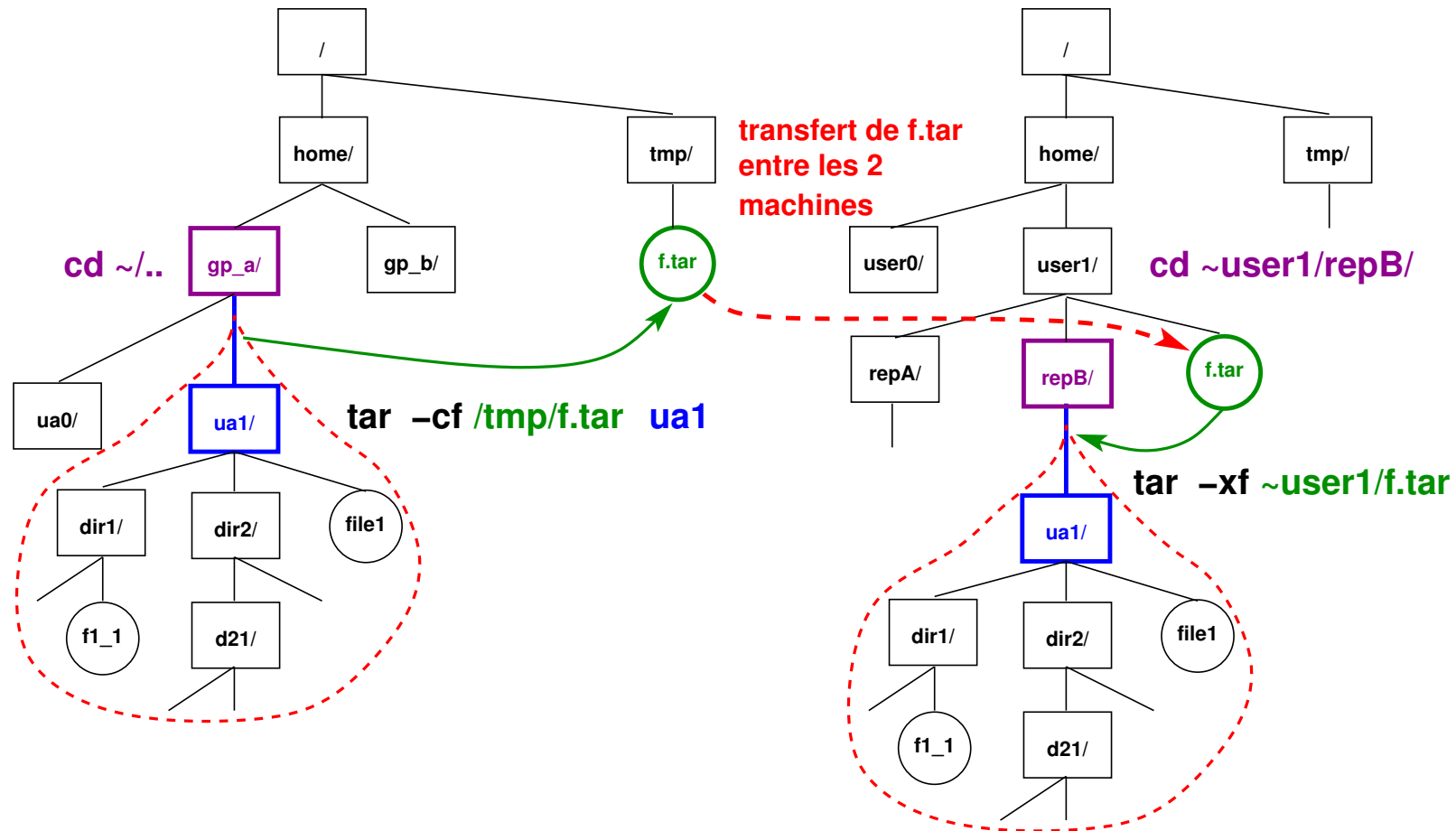


FIGURE 7 – **Transfert de branche via `tar`** : **création** de l'archive `f.tar`, **transfert** de l'archive entre les machines, puis **extraction** sous `repB`

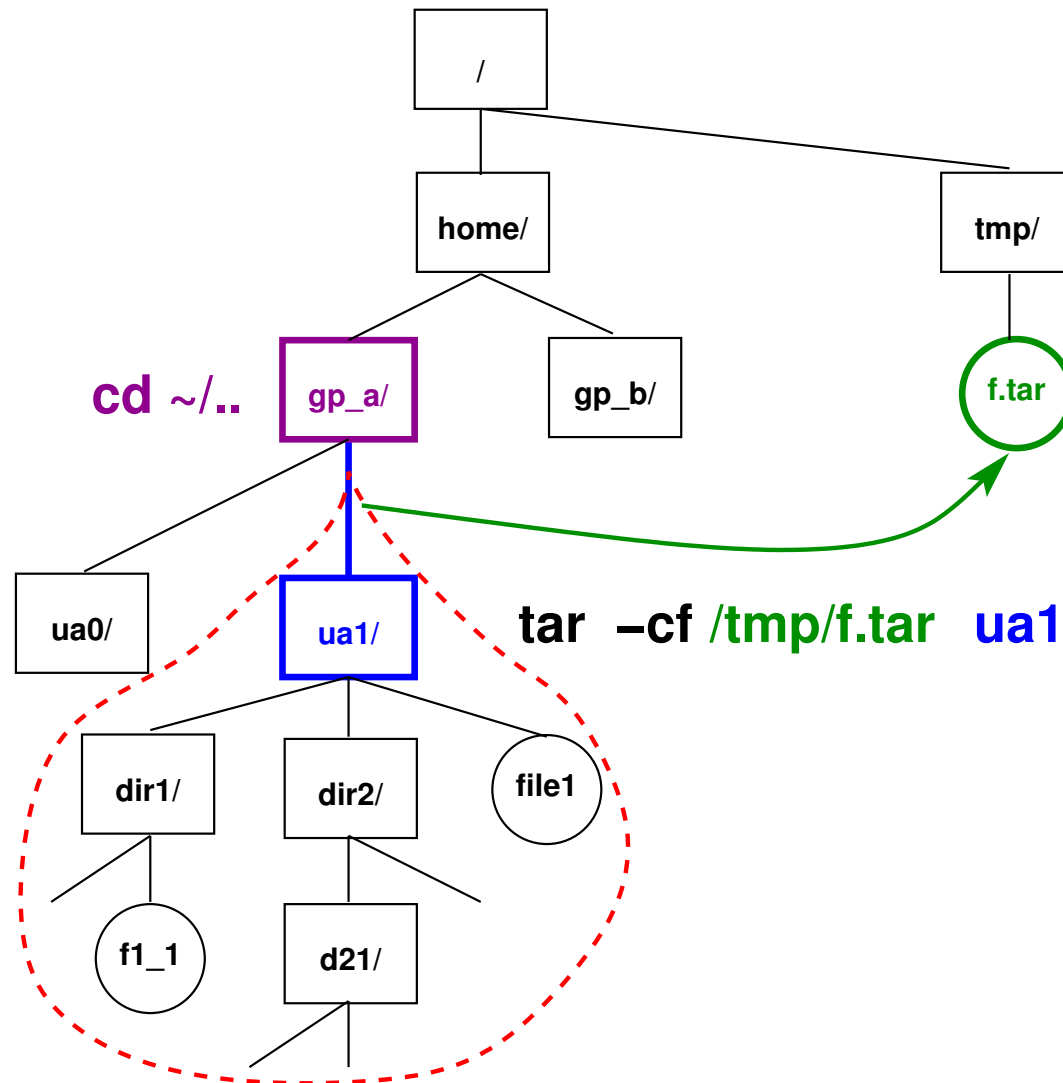


FIGURE 8 – Création

(sous `/tmp`) de l'archive `f.tar` de la branche de l'utilisateur

`ua1` :

1) `cd ~/..`

se placer au dessus

2) `tar -cf /tmp/f.tar ua1`

archiver `ua1`

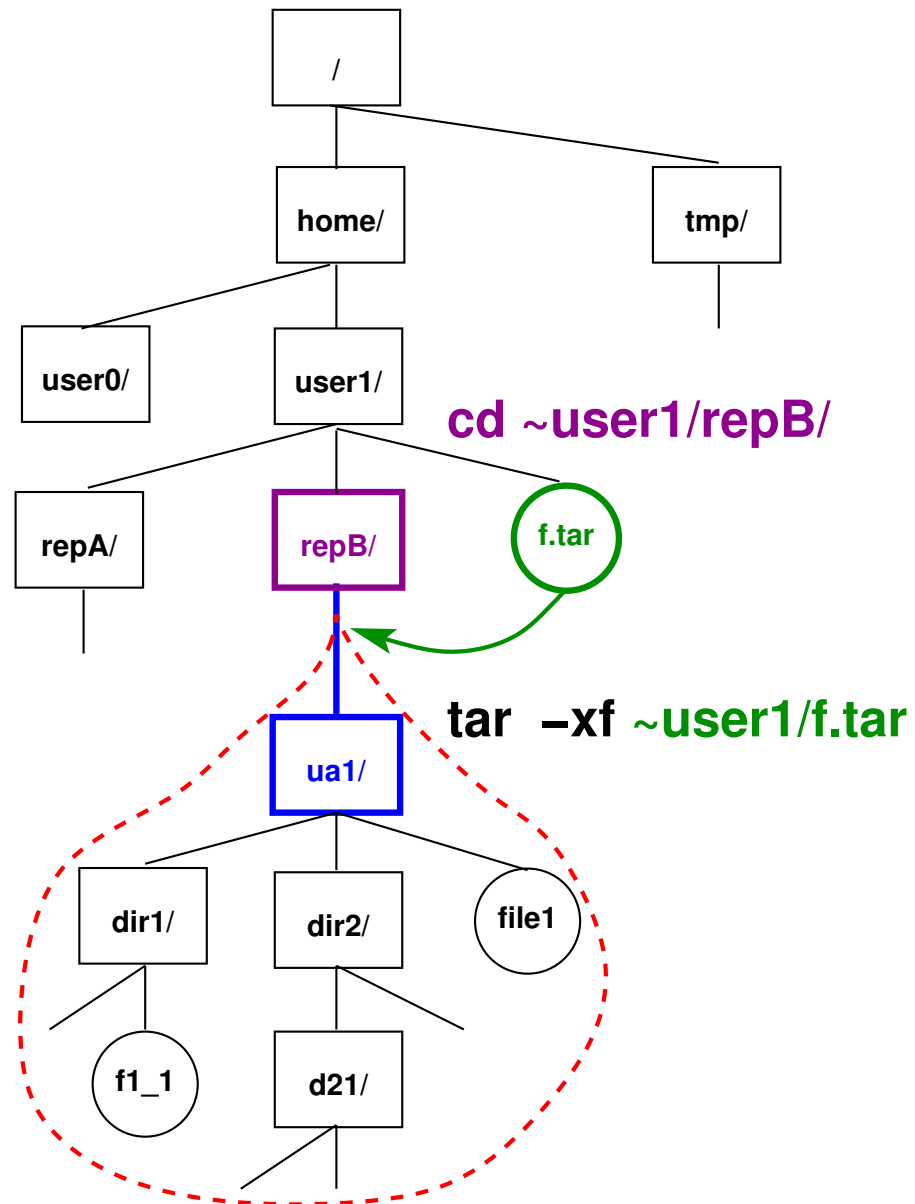


FIGURE 9 – **Restauration** de
 branche à partir de l'archive :
 1) `cd ~/repB/`
 on restaure juste en dessous
 2) `tar -xf ~/f.tar`

Exemples

```
cd ~/. ; tar -cvf /tmp/archive.tar user
```

archive toute l'arborescence de l'utilisateur `user` dans `archive.tar`
(se placer un niveau au-dessus de la branche à archiver)

```
tar -tf /tmp/archive.tar
```

affiche la liste des fichiers archivés dans `archive.tar`

```
tar -xvf /tmp/archive.tar
```

restaure toujours l'arborescence dans le répertoire **courant** (à partir de l'archive)
(se placer au niveau où « greffer » la branche à restaurer)

NB : l'option **-f** avec argument `-f fichier_archive` est obligatoire.



éviter les chemins absolus dans les sauvegardes,

sinon les fichiers seront obligatoirement restaurés au même endroit.

7.3 Copies et synchronisation de fichiers avec `rsync`

rsync [options] *source* [*user@host:*]*dest*

rsync [options] [*user@host:*]*source* *dest*

outil de copie plus puissant que `scp` ⇒ synchronisation de répertoires

— sur une même machine ou à distance

— très rapide car ne transmet que les différences et compression possible

— nombreuses options pour les sauvegardes et les miroirs

-r récursif **-v** (*verbose*) prolix **-z** (*zip*) compression avant transfert

-u (*update*) mise à jour : ne transmet que les fichiers plus récents côté source

-t, **-p** conserve la date (*time*), les droits (*permissions*)

--exclude=*motif* sauf les fichiers répondant au motif indiqué

-n essai à vide avant d'activer

Remarque : rôle des / terminaux

`rsync -r rep1 user@host:/tmp` crée le répertoire `/tmp/rep1/` sur la machine distante et y recopie récursivement le contenu de `rep1`

`rsync -r rep1/ user@host:/tmp` recopie récursivement le contenu de `rep1/` dans `/tmp/` sur la machine distante sans y créer de niveau `rep1`

`rsync -r rep1 user@host:/tmp` est donc équivalent à

`rsync -r rep1/ user@host:/tmp/rep1/`

Exemple

```
rsync -rvtpu --exclude='*~' \  
    user@sappli1.datacenter.dsi.upmc.fr:mni/unix/ ~/unix-mni
```

met à jour (u) récursivement (r) le répertoire local `~/unix-mni` à partir du répertoire `~user/mni/unix/` du serveur en conservant droits (p) et dates (t), mais sans transférer les fichiers de sauvegarde de suffixe `~` (`--exclude`)

8 Droits d'accès aux fichiers

Chaque fichier a des permissions propres :

— trois types de permissions

r	<i>read</i>	lecture
w	<i>write</i>	écriture
x	<i>execute</i>	exécution
-		droit refusé

— pour trois «publics»

type	propriétaire	groupe	autres
-/d/l	user	group	others
-	r w x	r w x	r w x

— donc 3x3 = 9 caractères + un caractère à gauche pour le type du fichier

- = fichier ordinaire, **d** = répertoire, **l** = lien symbolique

8.1 Affichage des droits d'accès avec `ls -l`

Exemple : `ls -l ~lefrere/M1/Config/`

```
drwxr-xr-x 2 lefrere ens 1024 sep 17 2009 lisp
-rwxr-xr-x 1 lefrere ens 1076 oct 7 2009 MNI.bash_profile
-rwxr-xr-x 1 lefrere ens 3101 oct 22 2009 MNI.bashrc
lrwxrwxrwx 1 lefrere ens 15 sep 15 17:40 motd -> motd.16sept
-rw-r--r-- 1 lefrere ens 434 sep 15 21:18 motd.16sept
```

Première colonne :

- **d** si répertoire : signification particulière des droits
- **l** si lien (*link*) symbolique (raccourci vers **->**) :
c'est la cible qui porte les restrictions d'accès



La permission **w** pour quelqu'un d'autre que le propriétaire est à éviter.

8.2 Changement des droits d'accès avec `chmod`

`chmod` *mode* *liste_de_fichiers* où *mode* représente~:

- la portée : **u**, (*user*), **g**, (*group*), **o**, (*others*) ou **a** (*all*),
- suivie de **=** (définit les droits), **+** (ajoute un droit), ou **-** (enlève un droit),
- suivi de la permission **r**, **w**, ou **x**.

Exemple 1 : **`chmod go-r fichier`**

supprime les droits de lecture au groupe et aux autres

Exemple 2 : **`chmod u+w,go-w fichier`**

donne le droit d'écriture au propriétaire et le supprime au groupe et aux autres

Application : restaurer des droits raisonnables sur des fichiers issus de systèmes de fichiers non unix (clefs USB formatées sous windows)

9 Édition de fichiers textes

9.1 Les éditeurs sous unix et leurs modes

9.1.1 Éditeurs sous unix

- en mode texte : (nécessitent une connaissance du terminal utilisé)
 - **vi** sur-couche de `ex`, très puissant, présent sur tous les unix, version **vim** sous linux, éditeur sensible au langage (C, fortran, latex, ...) avec mise en valeur de la syntaxe par des couleurs
 - **emacs** encore plus puissant, mais plus gourmand en ressources
- en environnement graphique multifenêtres, avec menus, gestion de la souris, ...
xemacs, gvim, gedit, kwrite...



attention au codage des fichiers texte : ASCII / ISO-8859-1 / UTF-8

⇒ transcodage au vol par l'éditeur dans certains cas

9.1.2 Les modes des éditeurs

Deux modes principaux :

- **commande** : les caractères saisis sont **interprétés** comme des ordres (requêtes)
⇒ immédiatement exécutés
- **insertion** : les caractères saisis sont directement **insérés** dans le fichier.

Le mode **par défaut** est :

- le mode **commande** sous **vi** ⇒ déroutant au premier abord
passage en mode insertion par une requête
- le mode insertion sous **emacs**
requêtes introduites par des caractères de contrôle : `Ctrl`, `Échap`
exemple : `^X^C` pour terminer l'édition emacs

9.2 Principes de l'éditeur `vi`

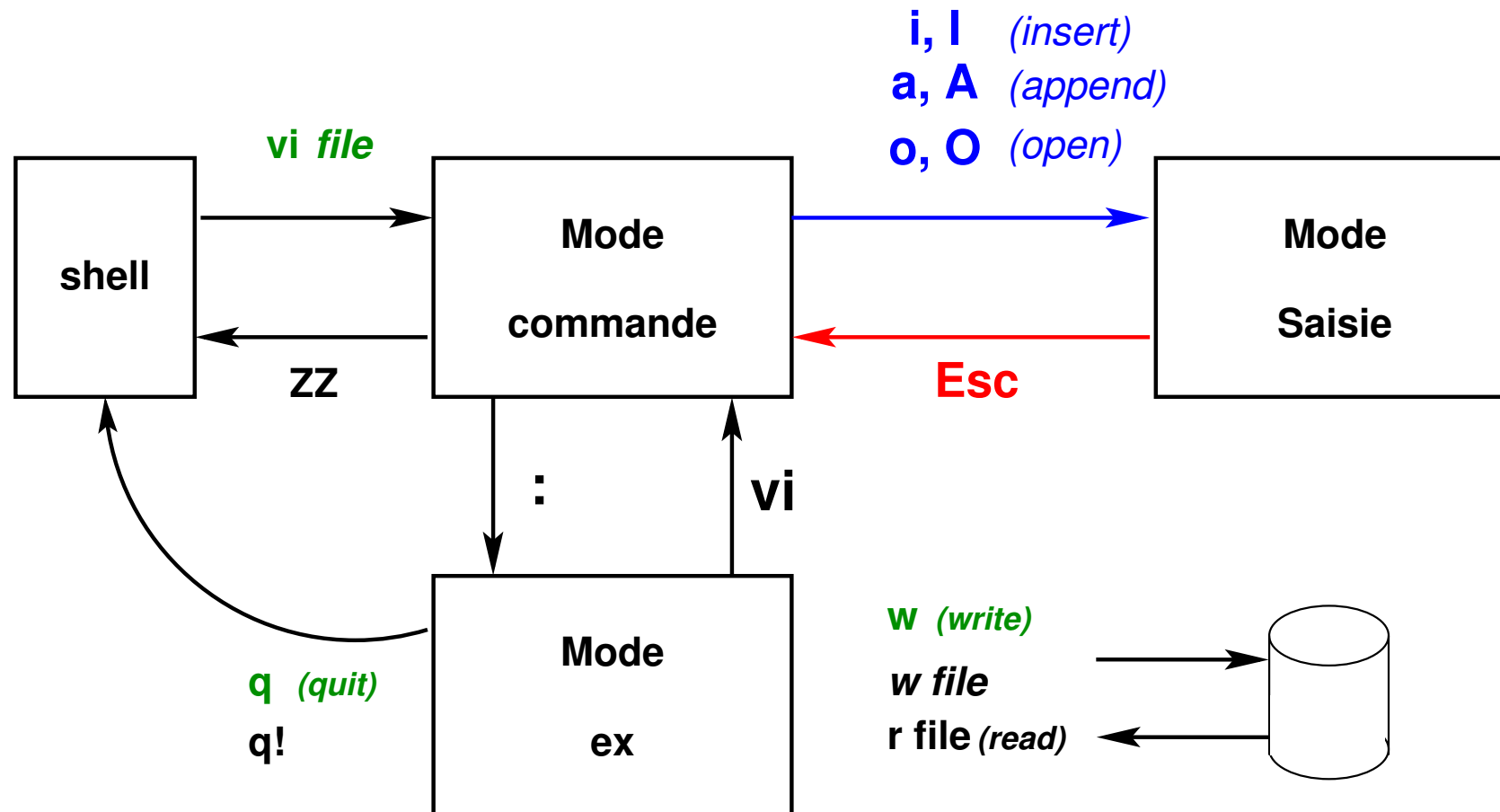


FIGURE 10 – Modes de fonctionnement de `vi`

- plusieurs requêtes pour **passer en mode saisie** :
 - **a** et **A** (*append*) ajout,
 - **i** et **I** (*insert*) insertion,
 - **o** et **O** (*open*) ouverture de ligne.
- une seule méthode pour **revenir au mode commande** :
touche d'échappement **Échap** (*escape*)

Un troisième mode, le mode **dialogue** est accessible de façon temporaire (affichage sur la ligne d'état en bas du terminal) depuis le mode commande de vi pour :

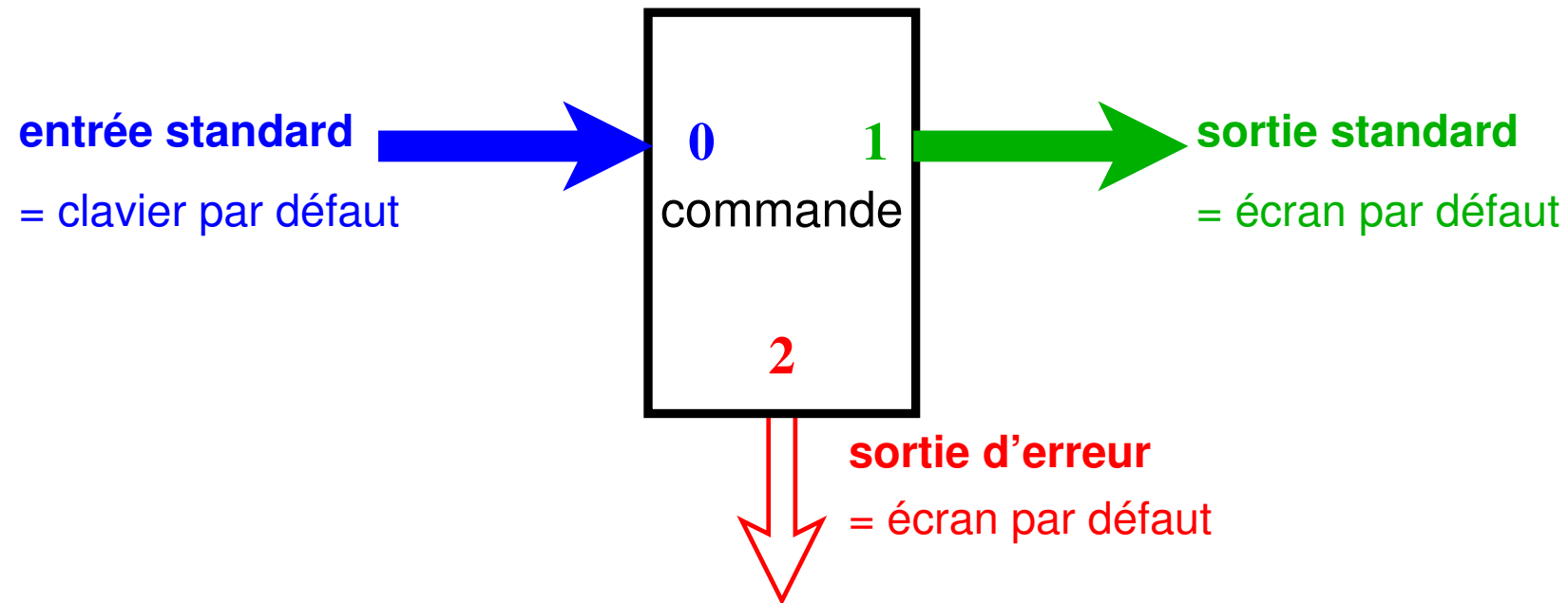
- passer des requêtes `ex` via :
- rechercher de motifs dans le texte (via **/** ou **?**)

Requête activée par `Entrée`,
puis retour immédiat en mode commande.

10 Redirections et tubes

10.1 Flux standard

Commande UNIX \Rightarrow trois flux standard de données :



10.2 Redirections

- Au lieu d'une saisie au clavier et d'un affichage à l'écran,
stocker de façon permanente l'information d'entrée ou de sortie
⇒ **rediriger** les flux standards à partir ou vers des **fichiers**
 - Combiner des commandes de base pour effectuer des traitements complexes
⇒ **rediriger** les flux standards vers les entrées/sorties d'**autres commandes**.
(mécanisme des tubes ou pipe-lines)
- ⇒ grande souplesse et puissance du système UNIX

10.2.1 Redirection de sortie vers un fichier (> et >>)

syntaxe

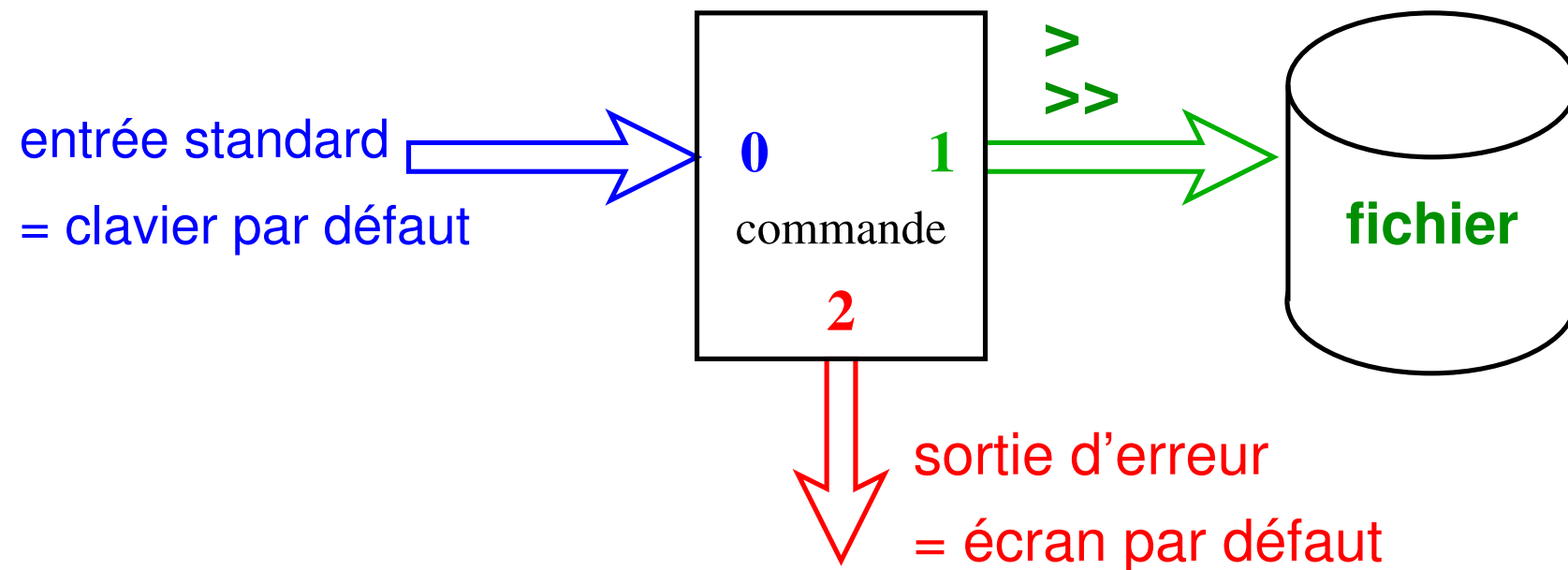
commande > *fichier*

syntaxe

commande >> *fichier*

Le fichier résultat est créé.

pour ajouter le résultat à la fin du fichier



Exemples

liste détaillée dans un fichier

```
ls -l > liste.txt
```

10 premières puis 10 dernières lignes

```
head fic.txt > deb+fin
```

```
tail fic.txt >> deb+fin
```

les noms des fichiers sources fortran,
puis ceux des fichiers en C

```
ls *.f90 > liste_f+c
```

```
ls *.c >> liste_f+c
```

Attention : le shell interprète très tôt les redirections

⇒ ne pas rediriger la sortie vers le fichier d'entrée



```
cat -n fic1 > fic1      efface le contenu du fichier fic1
```

Solution : `cat -n fic1 > tmp ; mv tmp fic1`

10.2.2 Redirection de l'entrée depuis un fichier (<)

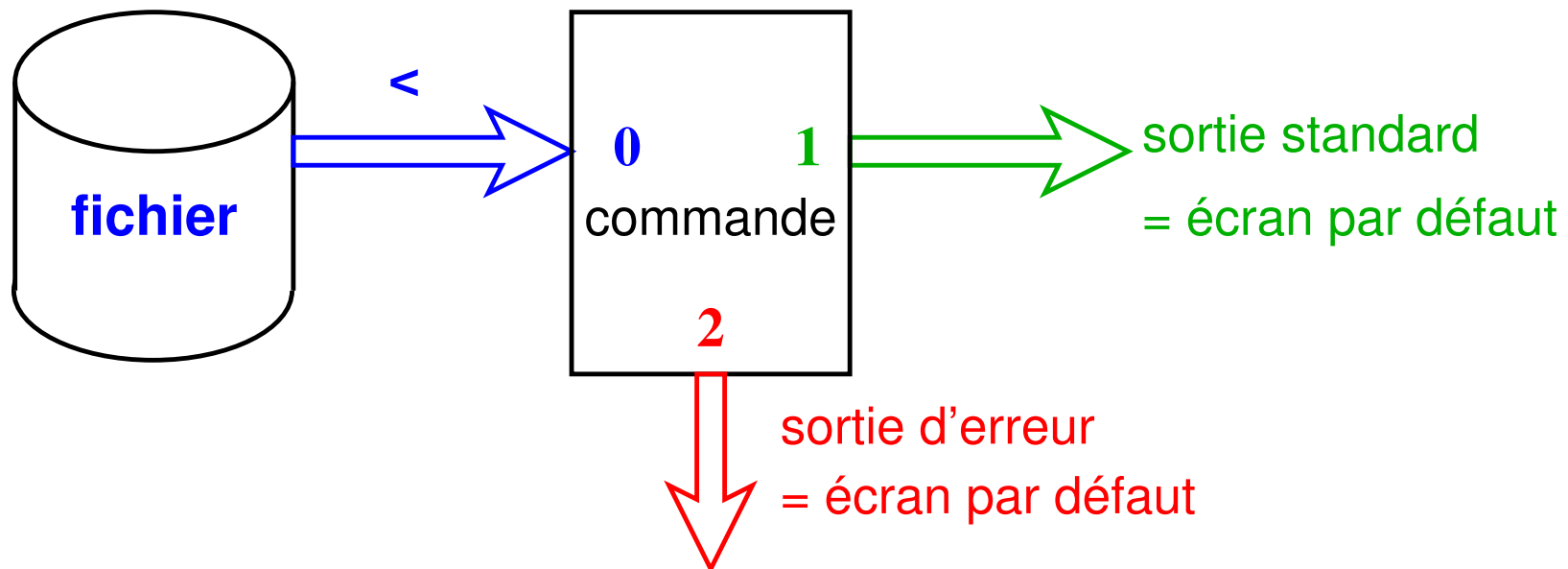
syntaxe

commande < *fichier*

Exemple : lecture des données d'entrée d'un exécutable sur un fichier au lieu de la saisie au clavier

le fichier doit exister au préalable.

a.out < entrees



10.3 Tubes ou *pipes* (|)

Appliquer deux traitements successifs à un flux de données :

- Méthode **séquentielle** avec fichier intermédiaire :

```
commande_1 > fichier
```

=> attente éventuelle

```
commande_2 < fichier
```

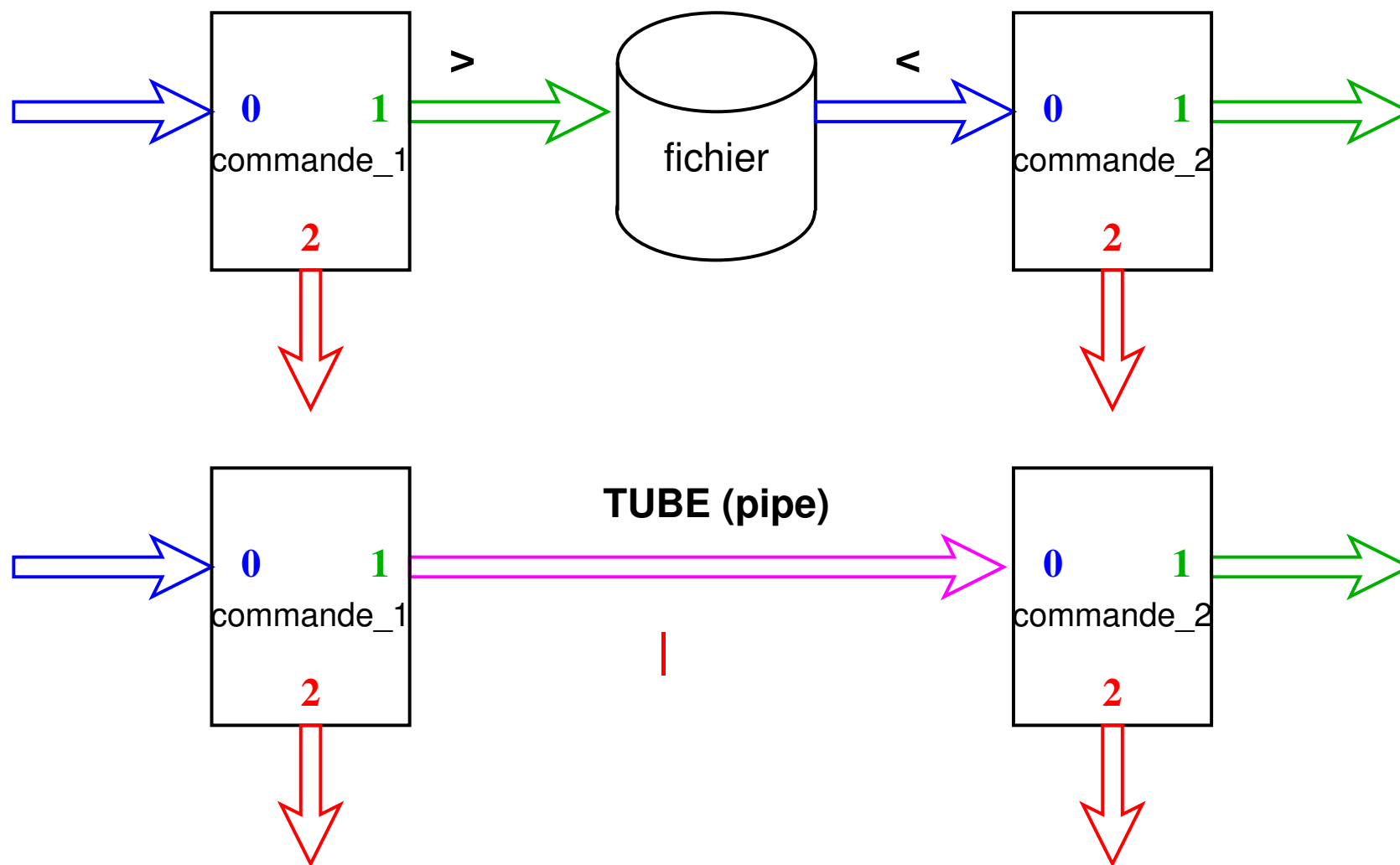
```
rm fichier
```

- **Traitement à la chaîne** en connectant les deux processus par un **tube** ou *pipe* = zone mémoire ⇒ communication synchronisée entre les 2 processus

syntaxe

```
commande_1 | commande_2
```

plus **rapide** que le traitement séquentiel



Exemple 1 : affichage paginé de la liste des fichiers du répertoire courant

Méthode séquentielle (à éviter)

```
ls -l > liste
more liste
rm liste
```

Chaînage avec tube (à préférer)

```
ls -l | more
```

Exemple 2 : affichage de la 12^e ligne du fichier `toto`

Méthode séquentielle (à éviter)

```
head -n 12 toto > tmp1
tail -n 1 tmp1
rm tmp1
```

Chaînage avec tube (à préférer)


```
head -n 12 toto | tail -n 1
```

Cas de plusieurs redirections

L'ordre des redirections sur la ligne est indifférent (avec une seule commande)

```
commande < entree > sortie
```

```
commande > sortie < entree
```

 Avec un tube, **ne pas détourner le flux** : pas de redirection sur des fichiers en sortie de la première commande ni en entrée de la seconde

```
commande_1 < entree | commande_2 > sortie
```

```
commande_1 > sortie | commande_2 < entree
```

10.4 Compléments

10.4.1 Redirection de la sortie d'erreurs vers un fichier (2> et 2>>)

_____ **syntaxe** _____
commande **2>** *fichier*

_____ **syntaxe** _____
commande **2>>** *fichier*



Attention : pas d'espace entre **2** et **>** pour ajouter les erreurs à la fin du fichier.

Exemple : stockage des diagnostics d'une compilation dans un fichier pour éviter le défilement à l'écran (afin de localiser d'abord la première erreur)

```
gfortran essai.f90 2> erreurs
```

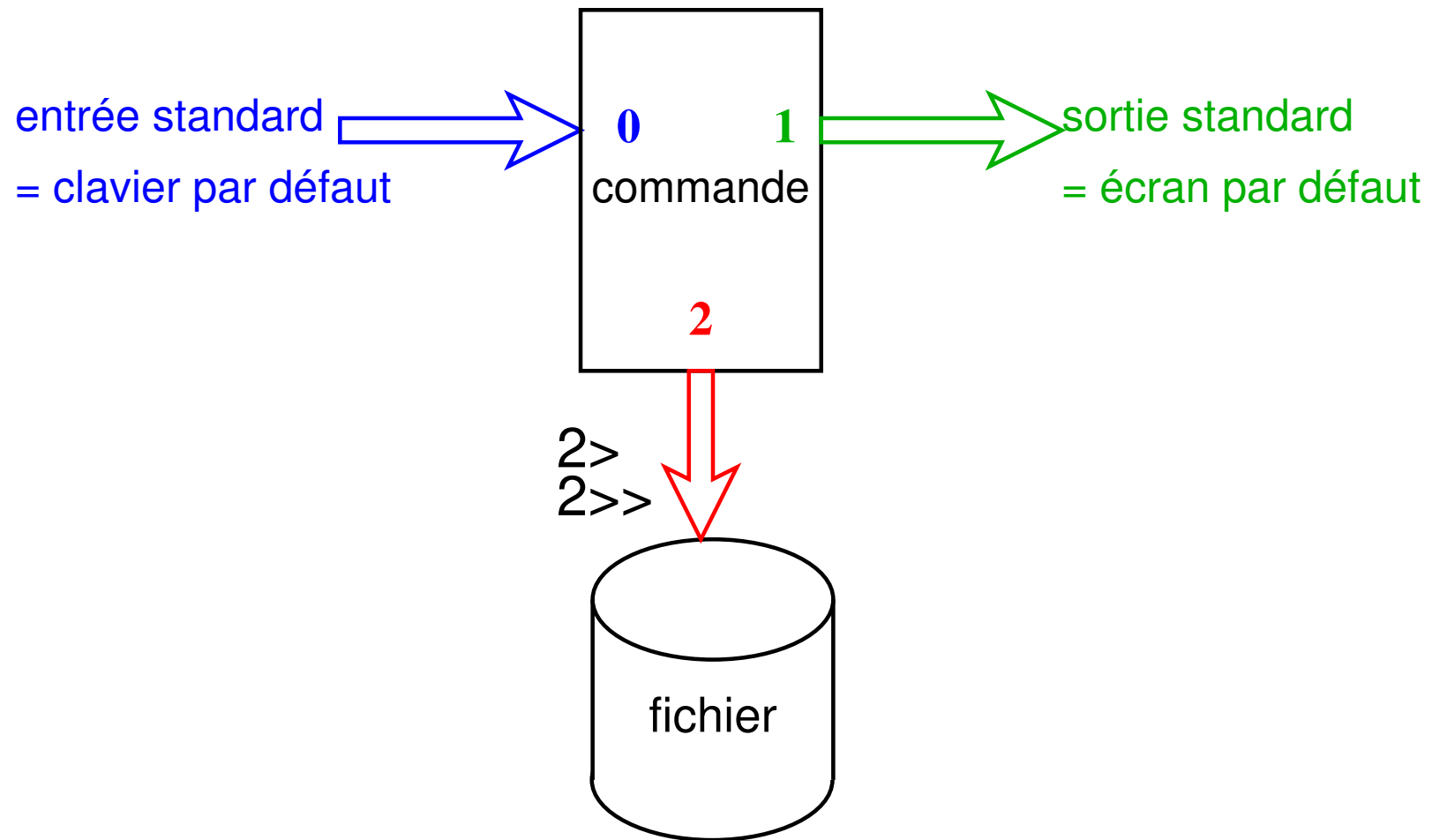


FIGURE 11 – Redirection de l'erreur

10.4.2 Redirection de l'erreur standard vers la sortie standard (2>&1)

Regroupement dans un même flux de la sortie standard et de la sortie d'erreur :

commande 2>&1 syntaxe

Exemple (on suppose que /etc/motd est accessible) :

```
cat /etc/motd /fichier_inexistant
```

affiche le mot du jour et un message d'erreur

```
cat /etc/motd /fichier_inexistant > resultat
```

affiche un message d'erreur

```
cat /etc/motd /fichier_inexistant > resultat 2>&1
```

n'affiche plus rien

N.-B. : la redirection de la sortie standard dans la dernière commande doit *précéder* la redirection de l'erreur standard vers le flux de la sortie standard.

10.4.3 Les fichiers spéciaux : exemple `/dev/null`

Répertoire `/dev` : *fichiers spéciaux* gérant des flux de données entre le calculateur et les périphériques (*devices*) : terminaux, imprimantes, disques, ...

`tty` affiche le nom du fichier spécial particulier attribué à un terminal
le fichier spécial `/dev/tty` désigne de façon générique le terminal attaché à la connexion.

`/dev/null` = fichier spécial « poubelle » (vide) ou trou noir !
⇒ utilisé pour se débarrasser de certaines sorties inutiles.

commande `2> /dev/null`

empêche le flux d'erreur de s'afficher à l'écran.

```
find rep -name "nom" -print 2> /dev/null
```

évite l'affichage des messages d'erreur

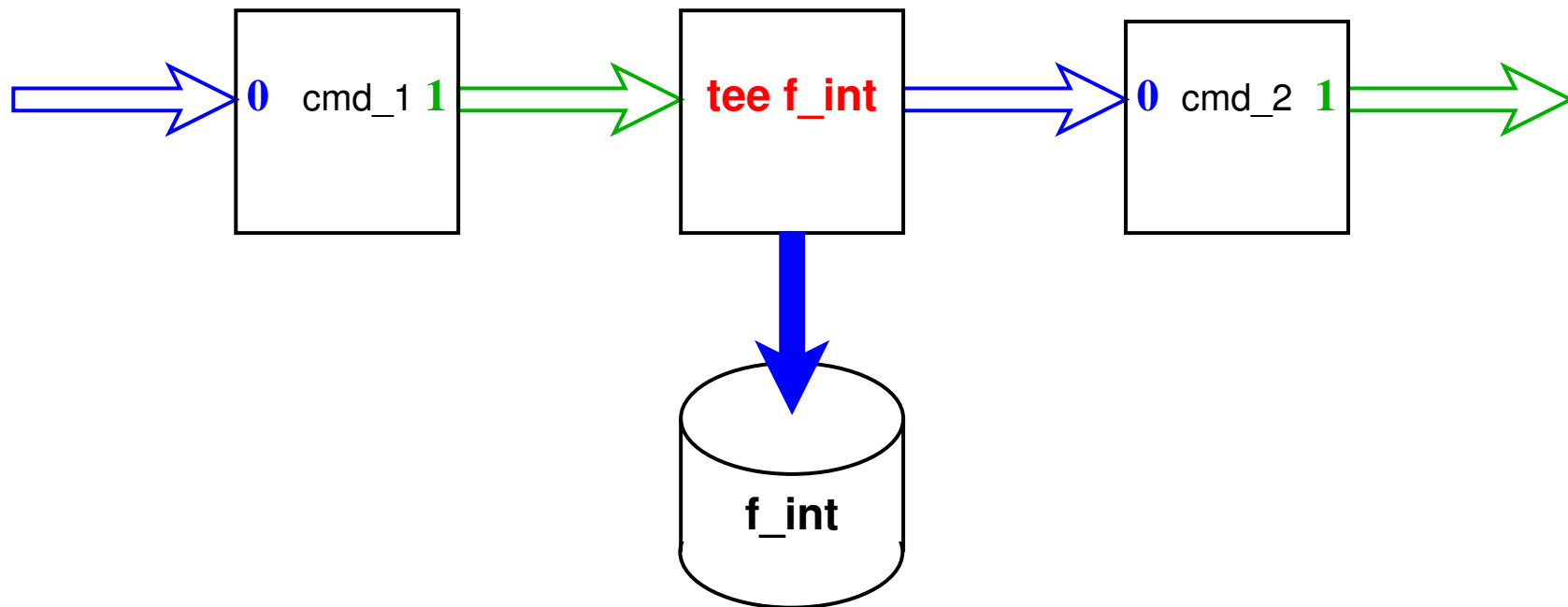
quand on tente d'accéder à des fichiers non autorisés.

10.4.4 Duplication de flux : tee

tee (**T** en anglais) **duplique** le flux de son entrée standard vers (a) le fichier passé en argument et (b) sa sortie standard.

Application : conserver une trace du résultat intermédiaire d'un tube :

```
cmd_1 | tee f_intermediaire | cmd_2
```



11 Filtres élémentaires

11.1 Définition

filtre = commande qui lit l'entrée standard , effectue des transformations sur ces données et affiche le résultat sur la sortie standard .

Exemples de filtres : `cat`, `wc`, `tail`, `head`, `tr`, `sort`, `grep`, `sed`, `awk`...
mais `ls`, `date`, `vi`... ne sont pas des filtres.

Utilisations

- **filtre** + saisie des données d'entrée au clavier et `^D` pour terminer
- **filtre** < *fic* lit dans un fichier par redirection d'entrée
- mais beaucoup de filtres (sauf `tr`) admettent aussi comme paramètres un nom de fichier ou une liste de fichiers d'entrée :

```
filtre fic1 fic2 ...
```

équivalent (presque car on perd ici les noms des fichiers) à :

```
cat fic1 fic2 ... | filtre
```


11.2 Classement avec `sort`

sort trie , regroupe ou compare toutes les **lignes** des fichiers passés en paramètre

Par défaut : ordre **lexicographique** sur tous les champs de la ligne

⇒ sensible aux variables de langue (`LANG` et `LC_...`)

Options :

-r (*reverse*) pour trier selon l'**ordre inverse**

-f pour **ignorer la casse** (majuscule/minuscule)

-n (*numeric*) pour trier selon l'**ordre numérique croissant**

séparateur décimal `.` ou `,` selon `LC_NUMERIC`

-u (*unique*) pour fusionner les lignes ex-æquo

-k *début* [*,* *fin*] classement **selon les champs** (*key*)

de numéros entre *début* (et *fin*)

-t *délim* choisit le séparateur de champs *délim* (blanc par défaut)

-b (*blank*) pour ignorer les blancs en tête de champ (cas de blancs multiples)

Exemples

```
sort /etc/passwd
```

classe les lignes du fichier `/etc/passwd` par ordre lexicographique

```
ls -l | sort -k8,8*
```

classe les fichiers par ordre lexicographique

```
ls -l | sort -k8,8*r
```

classe les fichiers par ordre lexicographique inverse

* : date et heure sont formatées en seulement deux champs (mandriva, mais pas `sapli1` par défaut)

```
ls -l | sort -k5,5n
```

classe les fichiers par ordre de taille croissante

```
ls -l | sort -k5,5n -u
```

idem, mais n'affiche qu'un exemplaire pour une taille donnée (fusion des ex-æquo)

```
wc -l * | sort -k1,1n
```

classe les fichiers par nombre de lignes croissant

```
sort -t: -k3,3n /etc/passwd
```

classe les lignes du fichier `passwd` (séparateur de champ `:`) par numéro d'id croissant (champ 3)

11.3 Transcription avec `tr`

`tr jeu1 jeu2` substitue à chaque caractère de l'ensemble fourni en premier paramètre son correspondant pris dans le deuxième ensemble.

`tr '123' 'abc'` change les **1** en **a**, les **2** en **b** et les **3** en **c**.

`tr -d jeu` pour supprimer (*delete*) les caractères d'un ensemble

`tr -s jeu1 jeu2` (*squeeze-repeats*) pour supprimer les répétitions de caractères de l'ensemble **d'arrivée** (après substitution)

Restrictions



Filtre **pur** : n'admet pas de nom de fichier en paramètre ⇒ redirections.

Travaille octet par octet ⇒ pas encore compatible UTF-8

⇒ ne tient pas compte du contexte (ne traite pas des motifs)

⇒ séquentiel sans mémoire (avantage !)

Compléments

Peut utiliser les séquences de contrôle de l'ascii (`\r` =CR, `\n`= NL)

`tr -d '\r'` supprime les « retour chariot » des fins de lignes (issus de windows)

Peut utiliser des intervalles (ascii)

`tr a-z A-Z`

ou mieux des classes de caractères (avec les accentués selon `LC_COLLATE`)

`tr '[:lower:]' '[:upper:]'` (en iso-latin, pas en UTF)

11.4 Autres filtres élémentaires

`head` / `tail` affiche les lignes du début (entête) / de la fin du fichier

`expand` / `unexpand -a` traduit les tabulations en espaces et inversement

`cut` sélectionne des colonnes (champs) dans un flux

12 Expressions régulières ou rationnelles

Recherche de chaînes de caractères qui satisfont à un certain motif (*pattern*)

⇒ **syntaxe** particulière pour décrire des **motifs génériques** :

une *expression rationnelle*

Expressions rationnelles utilisées par les éditeurs **ex**, **vi** et **sed**, les filtres **grep** et **awk**, ainsi que `perl`, `python`, `php`, `JavaScript`...

Deux versions exclusives de la syntaxe :

- expressions rationnelles de base **BRE** : **Basic Regular Expressions**
(`ex`, `vi`, `sed`, `grep`)
- expressions rationnelles étendues **ERE** : **Extended Regular Expressions**
(`awk`)

12.1 Signification des caractères spéciaux

- . (point) représente un caractère quelconque et un seul
- \ (contre-oblique : *backslash*) sert à protéger le caractère qui le suit pour empêcher qu'il ne soit interprété
- * (étoile) représente un nombre d'occurrences quelconque (**zéro**, une ou plusieurs occurrences) du caractère ou de la sous-expression qui précède



Ne pas confondre ces caractères spéciaux des expressions rationnelles avec les caractères génériques (*wildcards*) pour les noms de fichiers, * et ? qui sont, eux, interprétés par le shell.

Exemples

a* un nombre quelconque de fois le caractère a (y compris une chaîne vide)

a**a*** une ou plusieurs fois le caractère a

.* un nombre quelconque de caractères quelconques (y compris une chaîne vide)

.* au moins un caractère

\. un point suivi d'un caractère quelconque

***** un nombre quelconque (y compris zéro) de contre-obliques

12.2 Ancres

Les **ancres** (*anchor*) ne représentent aucune chaîne, mais permettent de spécifier qu'un motif est **situé en début ou en fin de ligne** :

^ (accent circonflexe : *caret*) spécial **en début** de motif, représente **le début de ligne**

\$ (dollar) spécial **en fin** de motif, représente **la fin de ligne**

^a une ligne commençant par un a

^a.*b\$ une ligne commençant par a et finissant par b

^\$ une ligne vide

^.*\$ une ligne quelconque, y compris vide

^.*\$ une ligne non vide

12.3 Ensembles de caractères

Un et un seul caractère choisi **parmi un ensemble** de caractères spécifiés entre crochets : `[ensemble_de_caractères]`

À l'intérieur d'un tel ensemble, les caractères spéciaux sont :

- utilisé pour définir des **intervalles** selon l'ordre lexicographique (dépend des variables de langue)
- ^ en tête pour spécifier le **complémentaire** de l'ensemble
-] qui délimite la **fin** de l'ensemble, sauf s'il est placé en première position

À l'intérieur des ces ensembles peuvent figurer des **classes de caractères**
`[:lower:], [:upper:], [:alpha:], [:digit:], [:alnum:]`

Exemples

<code>[a0+]</code>	un des trois caractères a, 0 ou +
<code>[a-z]</code>	une lettre minuscule
<code>[a-z ; ? !]</code>	une lettre minuscule ou une ponctuation double
<code>[0-9]</code>	un chiffre
<code>[^0-9]</code>	n'importe quel caractère qui n'est pas un chiffre
<code>[-]</code>	un crochet fermant] ou un signe moins –

Exemples avec une classe :

`[[:digit:]]` au lieu de `[0-9]`

`[-+.[[:digit:]]` pour un chiffre, un point ou signe + ou –

13 Le filtre `grep`

grep (*global regular expression print*)

affiche les lignes qui contiennent un motif passé en paramètre

_____ **syntaxe** _____

```
grep motif [liste_de_fichiers]
```

où *motif* est une expression régulière décrivant un motif générique

Principales options :

- i** ignore la casse (majuscule/minuscule)
- v** inverse la sélection (affiche les lignes sans le motif)
- l** affiche la liste des fichiers contenant le motif
- n** affiche les lignes contenant le motif précédées de leur numéro
- c** (*count*) affiche les noms des fichiers et le nbre de lignes qui contiennent le motif

Exemples :

```
grep lefrere /etc/passwd
```

affiche la ligne de cet utilisateur dans le fichier de mots de passe

```
grep '^!' test.f90
```

affiche les lignes commençant par ! dans test.f90 (commentaires)

```
grep '^ *!' test.f90
```

affiche les lignes dont le premier caractère non blanc est ! dans test.f90

```
grep -v '^ *!' test.f90
```

affiche les lignes qui ne sont pas des commentaires fortran

```
grep -v '^ *$' test.f90
```

affiche les lignes qui ne comportent pas que des blancs

```
ls -l | grep ^d
```

affiche la liste des sous-répertoires du répertoire courant avec leurs attributs



N.-B. : protéger les caractères spéciaux de l'interprétation par le shell, ici par des «`!`»

14 Le filtre sed

sed (*stream editor*) : éditeur de flux non interactif

filtre qui analyse ligne par ligne le flux d'entrée et le transforme selon des requêtes suivant une syntaxe similaire à celle de l'éditeur `ed`.

Deux syntaxes possibles suivant la complexité du traitement :

syntaxe

```
sed -e 'requête_sed' [liste_de_fichiers]
```

Les requêtes comportant des caractères spéciaux sont la plupart du temps protégées par des apostrophes de l'interprétation par le shell.

syntaxe

```
sed -f fichier_de_requêtes.sed [liste_de_fichiers]
```

où `fichier_de_requêtes.sed` contient des lignes de requêtes d'édition.

Autre option : **-n** n'affiche pas les lignes traitées (utiliser la requête `p`)

La plupart des requêtes sont adressables comme celles de `ex`.

Exemples : (**s** = *substitute*)

```
sed -e 's/ab/AB/'
```

change **le premier** ab de chaque ligne en AB

```
sed -e 's/10/20/g'
```

change **tous** les 10 en 20

```
sed -e '3,$s/0/1/g'
```

change **tous** les 0 en 1 à partir de la ligne 3

```
sed -e 's/00*/(&)/g'
```

entoure de parenthèses **tous** les groupes de 0

& représente le motif trouvé

```
sed -e 's/[0-9]/(&)/g'
```

insère des parenthèses autour de **tous** les chiffres

```
sed -e '/motif/s/0/1/g'
```

change **tous** les 0 en 1 dans les lignes

contenant `motif`

```
sed -e '/^#/d'
```

détruit les lignes commençant par #

```
sed -n -e '/^#/p'
```

affiche (*print*) les lignes commençant par #

-n pour éviter l'affichage par défaut de tout le fichier

15 Le filtre awk

awk : filtre programmable (agit ligne par ligne comme `grep`)

fonctionnalités de **calcul** de type **tableur**, syntaxe proche du langage C

⇒ sensible aux variables de langue (**LC_NUMERIC**) : virgule décimale ou point

Deux syntaxes, sur la ligne de commande ou dans un fichier :

_____ **syntaxe** _____

```
awk 'instructions_awk' liste_de_fichiers_de_donnees
```

dans ce cas protéger les instructions de l'interprétation par le shell

_____ **syntaxe** _____

```
awk -f fich_de_programme liste_fichiers_donnees
```

Autre option :

-F *délim* spécifie le séparateur de champs (blancs et tabulations par défaut)

15.1 Structure des données pour awk

Pour chaque ligne (*record*), les données sont découpées en champs (*field*) selon le séparateur **FS** (*field separator*) :

- **\$0** la ligne courante
- **NR** (*number of record*), son numéro d'enregistrement (de ligne)
- **NF** (*number of fields*), son nombre de champs
- **\$1**, **\$2**, ... **\$NF** : son premier, deuxième, dernier champ

15.2 Structure d'un programme awk

Suite de couples : **sélecteur { action }**

⇒ Un **sélecteur** peut être :

- vide et il est vrai pour toutes les lignes
- une expression régulière étendue (ERE) entre **/** et **/**
le sélecteur est vrai si le motif est présent dans la ligne

- une expression logique évaluée pour chaque ligne
- une combinaison logique (via **&&**, **||** ou **!**) de sélecteurs
- un intervalle de lignes sous la forme : sélecteur1, sélecteur2
- **BEGIN** ou **END** qui introduisent des actions exécutées avant ou après la lecture des données

⇒ Une **action** est une suite d'instructions (affectations de variables, calculs, opérations sur des chaînes de caractères, ...) exprimées dans une syntaxe analogue à celle du langage C (structures de contrôle en particulier).

- Constantes chaînes de caractères entre « " ».
- Variables non déclarées et typées seulement lors de leur affectation
- Nombreuses fonctions, notamment numériques (`log`, `cos`, `int`, ...) et chaînes de caractères (`length`, `tolower`, ...) disponibles.

L'action s'applique séquentiellement à toutes les lignes sélectionnées

⇒ **pas de boucle explicite sur les lignes**

N.-B. : `awk` mal adapté s'il faut plusieurs lectures des données
(ex. : calcul de pourcentage)

15.3 Exemples de programmes `awk`

— affichage des lignes ayant la valeur **numérique** `2004` pour premier champ

`$1 == 2004 {print $0}` mais protéger du shell si hors fichier

`awk '$1 == 2004 {print $0}' fichier`

— affichage des lignes ayant la **chaîne** `toto` pour deuxième champ

`awk '$2 == "toto" fic` ne pas oublier les guillemets pour la chaîne

— affichage des lignes avec leur numéro (équivalent de `cat -n`)

`awk '{print NR, $0}' fichier`

— échange des champs 1 et 2 et affichage :

`awk '{a=$1 ; $1=$2; $2=a; print $0}' fic`

— affichage du nombre de lignes du fichier (équivalent de `wc -l`)

`awk 'END {print NR}' fic`

- Calcul de la moyenne du champ 1 :

```
BEGIN{ n=0; s=0} (initialisation facultative)
{n++ ; s+=$1} (cumul)
END{ print "moyenne = ", s/n} (affichage)
```

- Calcul de la moyenne des valeurs supérieures à 10 du champ 1 :

```
BEGIN{ n=0; s=0} (initialisation facultative)
$1 > 10 {n++ ; s+=$1} (cumul conditionnel)
END{ if (n > 0 ) {
    print "moyenne = ", s/n (affichage)
    }
    else {
    print "pas de valeurs > 10"
    }
}
```

15.4 Mise en garde sur les caractères non-imprimables

Les **caractères de contrôle** dans les fichiers texte ne sont **pas toujours visibles** à l'affichage et l'édition.

— extrait de man ascii —

Oct	Dec	Hex	Car	
010	8	08	BS	'\b' (backspace)
011	9	09	HT	'\t' (horizontal tab)
012	10	0A	LF	'\n' (new line)
013	11	0B	VT	'\v' (vertical tab)
014	12	0C	FF	'\f' (form feed)
015	13	0D	CR	'\r' (carriage ret)

⇒ risques d'**erreur avec les filtres**.

- confusion entre espaces et **tabulations** : erreur sur motif `grep` ou `sed`
- fin de ligne avec **\r\n** (DOS) : erreur si ajout de caractères en fin de ligne
awk '{print \$0 "texte"}' ⇒ **texte** entre **\r** et **\n**
 ⇒ **texte** écrase le début de ligne

Comment visualiser ces caractères de contrôle ?

avec **cat** : `cat -A (All)` ou `cat -vET` affiche les retour chariot (**^M** avec **-v**),
les fins de ligne (**\$**, avec **-E**, (**End**)) et les tabulations (**^I** avec **-T**, (**Tab**)).

sous **vi** : option **:set list** \Rightarrow **^I** pour tabulation, **\$** pour fin de ligne
si `vi fic` affiche **[dos]** sur la ligne d'état, `fic` comporte des **\r**
`vi -b fic (binary)` affiche **^M** pour les retour chariot

avec **od** : `od -tc` affiche **\t** pour tabulation, **\r** retour chariot, **\n** retour ligne

Rappels : **expand** transforme les tabulations en espaces

tr comprend les séquences d'échappement comme **\t**, **\r**, **\n** et **\b**

16 Gestion des processus

16.1 Généralités : la commande `ps`

Processus = tâche élémentaire identifiée par un **numéro unique** ou *pid* (*process identifier*).

Afficher la liste des processus avec la commande **ps**

⇒ par défaut ceux de l'utilisateur et associés au même terminal

3 syntaxes pour sélectionner les processus et les informations affichées par **ps** :

System V, BSD, et Posix en cours d'implémentation (contrôler avec **man -a**).

Principales options :

-e (posix **-A**) affiche tous les processus de tous les utilisateurs

-U *user_list* sélectionne les processus appartenant à cette liste d'utilisateurs ou d'UID (séparés par des virgules sans espace)

-f (*full*) affiche une liste complète d'informations sur chaque processus

Exemples de sélection des processus

```
$ ps
```

PID	TTY	TIME	CMD
1212592	pts/2	0:00	ps
1294516	pts/2	0:01	bash

les processus de l'utilisateur sur le pseudo-terminal courant, affiché par `tty: /dev/pts/2`

```
$ ps -U lefrere
```

PID	TTY	TIME	CMD
307400	-	0:02	sshd
1212590	pts/2	0:00	ps
1294516	pts/2	0:01	bash
1294620	-	0:02	sshd
1294621	pts/6	0:01	bash

les processus de l'utilisateur `lefrere` sur toutes les consoles (ici **2** et **6**) accédant au serveur

N.-B. : la commande `ps` se voit agir.

Format de sortie de `ps`

Principaux champs affichés :

UID	PID	PPID	TTY	VSZ	CMD
n° utilisateur	n° du processus	n° du père	terminal	taille	commande

\$ ps -f (full)

```

      UID      PID      PPID      C      STIME      TTY      TIME  CMD
lefrere 1294516  307400    0 00:23:53 pts/2    0:01 -bash
lefrere 2027692 1294516   45 00:59:00 pts/2    0:00 ps -f

```

Affichage interactif des processus : commande **top** (**u user** pour sélectionner)



commande puissante mais consomme des ressources !

16.2 Caractères de contrôle et signaux

Caractères de contrôle (notés **^X** pour `Ctrl X`) interprétés par le shell

⇒ gestion des processus attachés au terminal et des flux d'entrées/sorties.

^? ou ^H	<code>erase</code>	effacement du dernier caractère
^L	<code>clear</code>	efface l'écran
^S	<code>stop</code>	blocage de l'affichage à l'écran
^Q	<code>start</code>	déblocage de l'affichage à l'écran
^D	<code>eof</code>	fermeture du flux d'entrée (fin de session en shell)
^C	<code>int</code>	interruption du processus
^Z	<code>susp</code>	suspension du processus en cours

stty gère l'affectation des caractères de contrôle à certaines fonctions

stty -a indique leur affectation courante (ex : `erase=^?; eof=^D`)

Un caractère de contrôle ne peut agir que sur le processus en interaction avec le terminal auquel il est attaché.

16.3 Commande `kill`

Intervenir sur un autre processus (ex. : application graphique qui ne répond plus)

⇒ le désigner par son numéro et lui envoyer un *signal*

kill *pid* où *pid* est le numéro du processus

kill envoie par défaut un signal de terminaison = **kill -s TERM**

si le processus ne s'interrompt pas, `kill -s KILL` (ou `kill -s 9`)

16.4 Processus en arrière plan

Système UNIX multi-tâche :

- commandes longues non-interactive en **arrière-plan** (*background*)
- « garder la main » pour d'autres commandes pendant cette tâche de fond (asynchrone)

syntaxe
`commande &`

Gestion des processus en arrière-plan :

- **jobs** affiche la liste des processus en arrière-plan avec leur numéro ($\neq pid$)
- **fg** (*foreground*) passe le job courant en premier plan
fg %num_job (passe le job num_job en premier plan)
- **bg** (*background*) passe le job courant en arrière-plan

Processus en arrière-plan \Rightarrow plus d'entrées au clavier

\Rightarrow redirections de l'entrée et de la sortie vers des fichiers

mais arrêté par la fermeture du terminal.

Exemples

- **xterm** en premier-plan \Rightarrow on « perd la main » dans la fenêtre initiale.
Dans la nouvelle fenêtre, terminer ce processus par `exit` ou `^D`
 \Rightarrow retrouver la main dans la fenêtre initiale.
- **xterm &** \Rightarrow conserve la main dans la fenêtre initiale.
Depuis la fenêtre initiale, terminer ce processus `xterm`
par `kill pid` ou par `fg` puis `^C`
- si on oublie le **&**, **^Z** pour suspendre le processus, puis **bg** pour le passer en arrière-plan

17 Code de retour d'une commande

17.1 Code de retour d'une commande (\$?)

Toute commande UNIX renvoie en fin d'exécution un code entier : valeur de retour (cf. `exit()` dans `main` en C) ou statut de fin (*return status*) accessible via **\$?**

Code de sortie = 0 \iff la commande s'est bien déroulée.

```
cd /bin
```

```
echo $? affiche 0
```

```
cd /introuvable affiche un message d'erreur
```

```
echo $? affiche 1
```

17.2 Combinaison de commandes (& &)

_____ syntaxe _____

```
commande_1 && commande_2
```

- La première commande est exécutée.
- Si et seulement si elle réussit (code de retour égal à zéro), la seconde est alors exécutée.

Par exemple, on lance un exécutable seulement si sa production (compilation et lien) s'est effectuée sans erreur.

```
gfortran source.f90 && a.out
```

18 Variables shell

Variables de l'interpréteur de commandes :

- non déclarées
- non typées a priori ⇒ **chaînes de caractères**
- pas d'héritage par les processus fils

18.1 Affectation et référence

- Syntaxe d'**affectation** (en shell de type BOURNE) :

_____ **syntaxe** _____

<i>variable= valeur</i> (sans espace autour du signe =)

- **Référence** à la valeur de la variable :

_____ **syntaxe** _____

<i>\$variable</i> ou, plus précisément <i>\${variable}</i>

La commande interne **set** (sans argument) affiche la liste des variables et leurs valeurs.

Exemples

```
alpha=toto ; b=35 ; c2=3b
```

```
echo alpha, b, c2 contiennent ${alpha}, ${b}, ${c2}
```

⇒ **alpha, b, c2 contiennent toto, 35, 3b**

```
set | grep alpha
```

⇒ **alpha=toto**

```
bb=${b}+${b} ; echo b vaut ${b}, bb vaut ${bb}
```

⇒ **b vaut 35, bb vaut 35+35**

 **pas d'arithmétique directement avec le shell** ⇒ utiliser **expr** ou **\$(())**

```
expr 3 + 12
```

affichent **15**

```
echo $( (3+12) )
```

18.2 Extension de la portée d'une variable : variables d'environnement

Exportation d'une variable vers les processus fils (shell de type Bourne) :

_____ **syntaxe** _____

```
export variable
```

Variables d'environnement systématiquement **héritées par les processus fils**.

Liste des variables d'environnement et de leur valeur : **env**

Variables d'environnement standard :

- **SHELL** : interpréteur de commandes utilisé (`bash`, `ksh`, `tcsh`, ...)
- **TERM** : type de terminal utilisé (`vt100`, `xterm`, ...)
- **HOME** : répertoire d'accueil
- **USER** : identifiant (nom) de l'utilisateur

- **PATH** : liste des chemins de recherche des commandes séparés par des « : »

Quand on lance une commande ou un exécutable :

- **avec /** dans le nom, on précise le **chemin d'accès explicite** :

par ex. `./a.out`

- **sans /** dans le nom, la **recherche** se fait dans tous les répertoires listés dans **PATH** en respectant l'ordre, par ex. `a.out`

Si on l'ajoute le point à la fin d'un `PATH` qui ne le contient pas, le répertoire courant est scruté en dernier : **`PATH="$PATH : ."`**



Ne pas placer le point au début du `PATH` pour des raisons de sécurité !



Attention : `PATH=""` \Rightarrow seules les commandes avec chemin sont trouvées
+ mémoire (cache) des chemins des commandes utilisées gérée par **hash**

18.3 Variables de localisation (langue, ...)

- **LANG**,
- **LC_ALL** qui résume les suivantes
- **LC_CTYPE** détermine la classification des caractères ([:lower:] par ex.)
- **LC_NUMERIC** (détermine le séparateur décimal : par ex. pour `sort`)
- **LC_COLLATE** (qui influe sur l'ordre lexicographique : important pour le classement avec `sort` et les expressions régulières avec des intervalles)
- **LC_TIME** pour la date et l'heure
- **LC_PAPER** A4 en Europe ou Letter aux États-Unis
- ...

Influent sur de nombreuses commandes (`date`, `man`, `wc`, `awk`, `sort`, `ls`, ...)

Valeurs affichées par la commande **locale**.

2 minuscules (langue) + `_` + 2 majuscules (variante locale) + `.` + nom du codage

Exemples : **C** (norme POSIX), **fr_FR.ISO-8859-1** ou **fr_CA.UTF-8**

19 Caractères interprétés par le shell

19.1 Substitution de commande

Résultat d'une commande (sa sortie standard) → chaîne de caractères stocké dans une variable ou repris comme argument d'une autre commande.

_____ **syntaxe** _____

\$ (commande)

 Ne pas confondre **$\${variable}$** et **$\$(commande)$** .

Utilisation :

- paramétrage de shell-scripts,
- calculs sur les entiers avec la commande **expr**

Exemples

```
qui=$(whoami)
```

affectation de la variable

```
echo ${qui}
```

puis affichage

```
echo je suis $(whoami)
```

utilisation directe

```
echo la date est $(date)
```

```
#
```

```
s1=$(expr 12 + 2)
```

calcul puis affectation à s1

```
echo la somme de 12 et 2 est ${s1}
```

⇒ *la somme de 12 et 2 est 14*

```
s2=$(expr $(expr 12 + 2) + 1) imbrication
```

19.2 Métacaractères du shell

<code>_</code> ou <code>TAB</code>	espace ou tabulation : séparateur (IFS)
<code>*</code> , <code>?</code> , <code>[...]</code>	constructeurs de noms de fichiers
<code>~</code>	répertoire d'accueil
<code><</code> , <code><<</code> , <code>></code> , <code>>></code> , <code> </code>	redirections
<code>\$</code> ou <code>\${...}</code>	évaluation de variable
<code>\$ (...)</code>	substitution de commande
<code>;</code>	séparation de commandes
<code>(...)</code> et <code>{...}</code>	groupements de commandes
<code> </code> et <code>&&</code>	associations de commandes
<code>!</code>	non logique
<code>&</code>	lancement en arrière plan
<code>#</code>	introduceur de commentaire

Les deux étapes d'interprétation : le shell, puis la commande

1. En premier lieu, **le shell** interprète la ligne de commande (espaces, caractères jokers, redirections, variables, ...)
2. Puis, **la commande** interprète certains caractères spéciaux pour elle-même. (expressions régulières pour `grep`, `sed`, ...)

Pour éviter d'exposer ces métacaractères à l'interprétation par le shell utiliser une des protections suivantes :

<code>\</code>	protection individuelle du caractère suivant (<i>backslash</i>)
<code>'...'</code>	protection forte (<i>quote</i>) : aucune interprétation
<code>"..."</code>	protection faible (<i>double quote</i>) : substitution de paramètres ou de commandes (<code>\$</code> interprété)

Exemples

— `grep '[0-9][0-9]*' fic`

affiche les lignes de `fic` comportant au moins un chiffre


— Affectation d'une chaîne comportant des blancs à une variable ou à un paramètre de commande : (nom de fichier avec des blancs)

`v1="avec blanc1" ; v2='avec blanc2' ; v3=avec\ blanc3`

— `echo ${TERM} \${TERM} "${TERM}" '${TERM}'`

⇒ `xterm ${TERM} xterm ${TERM}`

— `find ~lefrere -name '*.f90' -print`

 Sans protection, le shell remplacerait `*.f90` par la liste des fichiers de suffixe `.f90` dans le *répertoire courant* avant l'exécution de `find`

⇒ erreur de syntaxe s'il y en a plus d'un (**un seul** paramètre après `name`).

20 Shell-scripts

20.1 Fichiers de commandes ou shell-scripts

Fichier texte contenant des commandes, créé avec un éditeur de textes

Trois méthodes d'exécution :

1. **bash *fichier_de_cmdes*** éviter
2. puis rendre le script **exécutable** par **chmod +x *fichier_de_cmdes***
et donner le chemin d'accès du fichier de commandes
./fichier_de_cmdes faire suivre des paramètres éventuels
3. Ajouter dans **PATH** le chemin d'accès au fichier. Saisir
fichier_de_cmdes
qui est alors une commande recherchée dans l'ordre du PATH
⇒ éviter les noms des commandes existantes sur le système

En pratique, répertoire courant (**.**) à la fin du PATH,

mais mieux : scripts dans ***\${HOME}/bin/*** et ***\${HOME}/bin/*** dans le PATH

Exemple de shell-script sans paramètre

```
#!/bin/sh
#
# shell script sans paramètre
echo nous sommes le ; date
echo mon login est $(whoami)
echo "le calculateur est $(hostname) "
```

introduit les commentaires... sauf

sur la première ligne commençant par **#!**

⇒ **/bin/sh** précise le shell d'interprétation du script.

⇒ assure le portabilité du script

20.2 Les paramètres des scripts

Variables positionnées dans la procédure lors du lancement :

\$0

nom du fichier de commande (tel que spécifié lors de l'appel)

\$1, \$2, ... \$9

paramètres positionnels (arguments) avec lesquels la procédure a été appelée

le nombre de paramètres peut dépasser 9

⇒ accéder au dixième paramètre via `${10}`

\$*

chaîne formée par l'ensemble des paramètres d'appel "`$1 $2 $3 ...`"

\$#

nombre de paramètres positionnels lors de l'appel

\$\$

numéro du processus lancé (pid)

Exemples de procédure avec des paramètres

```
#!/bin/sh
# fichier proc0.sh
echo la procédure $0
echo a été appelée avec $# paramètres
echo le premier paramètre est $1
echo la liste des paramètres est $*
echo le numéro du processus lancé est $$
```

```
#!/bin/sh
# fichier concat.sh
# permet de concatener (cf "cat") deux fichiers ($1 et $2)
# dans le fichier $3
# en habillant le résultat avec le nom
# des fichiers initiaux en entete
#
ficin1=$1
ficin2=$2
ficout=$3
echo commande $0 avec les $# parametres $*
echo et le numero de processus $$
echo "début de la concatenation de\
  $ficin1 et $ficin2 sur $ficout"
echo '-----' > $ficout
```

```
echo \ | $ficin1 \ | >> $ficout
echo '-----' >> $ficout
cat $ficin1 >> $ficout
echo '-----' >> $ficout
echo \ | $ficin2 \ | >> $ficout
echo '-----' >> $ficout
cat $ficin2 >> $ficout
echo termine
exit 0
# fin du fichier concat.sh
```

20.3 Utilisation de la commande `set`

`set` suivie d'une option introduite par `-` (ou `+`) permet de positionner des réglages du shell ; les options suivantes sont utiles dans la phase de mise au point des procédures :

- `set -v` (*verbose*) affiche chaque commande (sans évaluation) avant de l'exécuter
- `set -x` (*xtrace*) affiche chaque commande (précédée du signe `+`) après évaluation des substitutions de commandes, `$(...)` et de variables, `${...}` avant de l'exécuter

Plusieurs possibilités lors de la mise au point :

- Placer la commande `set -vx` en tête du shell-script
- Ajouter ces options à la ligne `#!/bin/sh` \Rightarrow `#!/bin/sh -vx`
- Lancer le script avec `sh -vx shell-script`

21 Structures de contrôle en shell (sh)

21.1 Introduction

le shell = interpréteur de commandes

= langage de programmation

⇒ variables, paramètres des procédures

structures de contrôle

Pas de typage des variables ⇒ condition = code de retour des commandes

Mais souvent, code de retour de la commande **test**

Mots clefs réservés du shell : **if, then, else, fi, elif, for, do, ...**

Remarque : syntaxe différente en `csh`

⇒ préciser le shell dans les scripts par **#!/bin/sh** pour assurer la portabilité

21.2 Conditions

21.2.1 Structure `if ... fi`

syntaxe

```
if cmd  
then  
    commande (s)          (exécutées si le code de retour de cmd est 0)  
else  
    commande (s)          (exécutées si le code de retour de cmd est  $\neq$  0)  
fi
```

Partie **else** optionnelle.

Exemple avec la commande test

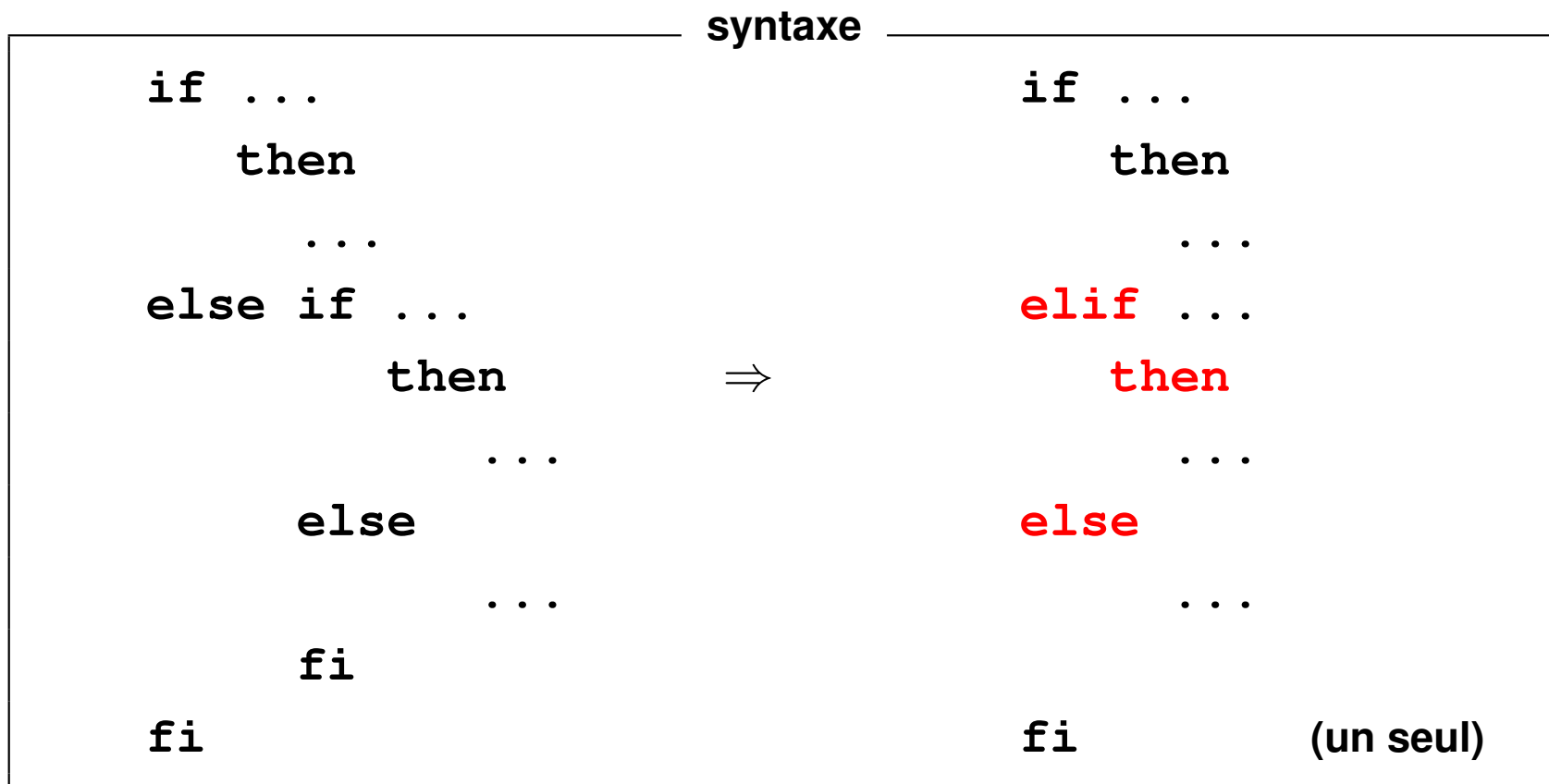
```
#!/bin/sh
if test $# -eq 0
then
    echo commande lancée sans paramètre
else
    echo commande lancée avec au moins un paramètre
fi
```

Exemple avec un tube

```
#!/bin/sh
# indique si l'utilisateur de nom $1 est connecté
if who | grep "^$1 "          code de retour = celui de grep
then                          (grep rend 0 si le motif est trouvé)
    echo $1 est connecté
fi
```

21.2.2 Structures `if` imbriquées : `elif`

Remplacer **else if** par **elif** \Rightarrow un seul **fi** (plus d'imbrication)



Exemple de `elif`

```
#!/bin/sh
if test $# -eq 0
then
    echo Relancer la cmde en ajoutant un paramètre
elif who | grep "^$1 " > /dev/null # sans affichage
then
    echo $1 est connecté
else
    echo $1 n'est pas connecté
fi
```

21.2.3 Énumération de motifs (cas) : `case ... esac`

— syntaxe —

```
case variable in  
    motif1) commande(s) ; ;  
    motif2) commande(s) ; ;  
    ...  
esac
```

La valeur de la variable est comparée avec les motifs successifs : à la première coïncidence, les commandes associées au motif sont exécutées jusqu'au `;;`, qui provoque la fin de l'exploration.

Syntaxe des motifs :

`*` = un nombre quelconque de caractères quelconques

`[xyz]` = l'un quelconque des caractères énumérés entre les crochets

`[x-z]` = l'un des caractères entre `x` et `z` dans l'ordre lexicographique

`motif1 | motif2 | motif3` = un quelconque des motifs séparés par des `|`

Exemple

```
#!/bin/sh
echo ecrivez OUI
read reponse
case ${reponse} in
    OUI)          echo bravo
                  echo merci infiniment      ;;
    [Oo][Uu][Ii]) echo merci beaucoup
                                                          ;;
    o*|O*)       echo un petit effort !      ;;
    n*|N*)       echo vous etes contrariant  ;;
    *)           echo ce n'est pas malin
                  echo recommencez          ;;
esac
```

Remarques

Les motifs peuvent se recouvrir, mais **seule** la première coïncidence provoque l'exécution de commandes

⇒ l'**ordre des motifs** est important.

En C-shell ou langage C, structure **switch** mais où chacun des motifs en coïncidence provoque l'exécution de commandes.

Structure **switch** équivalente à **case**

...**si** chaque cas est terminé par **breaksw / break;**

21.3 Les structures itératives

21.3.1 La structure `for ... do ... done`

_____ syntaxe _____

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

Liste des mots par défaut : les paramètres du script

```
"$@" (" $1 " " $2 " " $3 " ...)
```

Exemple avec liste explicite

```
#!/bin/sh
for mot in 1 5 10 2 "la fin"
do
    echo mot vaut ${mot}
done
```

⇒ boucle avec 5 passages

Exemple avec liste implicite

soit le script `do-echo.sh`

```
#!/bin/sh
for param
do
    echo +${param}+
done
```

`do-echo.sh 11 2 "3 3" 44`

affiche

```
+11+
+2+
+3 3+
+44+
```

_____ Liste générée par le joker * _____

```
#!/bin/sh
for fichier in *.f90
do
    echo fichier ${fichier}
done
```

_____ Procédure à un argument : le motif recherché _____

```
#!/bin/sh
motif=$1
for fic in $(grep -l ${motif} *)
do
    echo le fichier $fic contient le motif $motif
done
```

21.3.2 La structure `until ... do ... done` (*jusqu'à ce que*)

syntaxe

```
until commande  
do  
    commande (s)  
done
```

Les commandes entre `do` et `done` sont exécutées *jusqu'à ce que* la commande qui suit **`until`** rende un code nul.

Exemple Script qui boucle jusqu'à ce qu'un utilisateur se connecte :

```
#!/bin/sh
utilisateur=$1
until who | grep "^${utilisateur} " > /dev/null
do
    echo ${utilisateur} n'est pas connecté
    sleep 2
done
echo ${utilisateur} est connecté
exit 0
```

21.3.3 La structure `while ... do ... done` (*tant que*)

syntaxe

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

Les commandes entre `do` et `done` sont exécutées *tant que* la commande qui suit **while** rend un code nul.

Exemple Script qui boucle jusqu'à ce qu'un utilisateur se déconnecte :

```
#!/bin/sh
utilisateur=$1
while who | grep "^${utilisateur} " > /dev/null
do
    echo ${utilisateur} est connecté
    sleep 2
done
echo ${utilisateur} n'est pas connecté
exit 0
```

21.4 Compléments : branchements

21.4.1 La commande `exit`

exit [*statut_de_fin*] arrête l'exécution de la procédure et rend le *statut_de_fin* (0 par défaut) à l'appelant.

Utilisé pour arrêter le traitement en cas d'erreur après envoi d'un message

⇒ rendre alors un code $\neq 0$

...

```
if [ $# -lt 1 ]          # test sur le nb d'arguments
then
    echo "il manque les arguments" >&2
    # message sur sortie d'erreur
    exit 1 # sortie avec code d'erreur
fi
...
```

21.4.2 La commande **break**

break \Rightarrow sortie d'une boucle avant la fin ;

break n sort des n boucles les plus intérieures.

Nécessaire dans les boucles a priori infinies (`while true, until false`)
insérée dans un bloc conditionnel pour arrêter la boucle


```
#!/bin/sh
# fichier break.sh
while true          # boucle a priori infinie
do
    echo "entrer un chiffre (0 pour finir)"
    read i
    if [ "$i" -eq 0 ]
    then
        echo '**' sortie de boucle par break
        break          # sortie de boucle
    fi
    echo vous avez saisi $i
done
echo "fin du script"
exit 0
```

21.4.3 La commande `continue`

`continue` saute les commandes qui suivent dans la boucle et reprend l'exécution en début de boucle.

`continue` n sort des $n - 1$ boucles les plus intérieures et reprend au début de la n^{e} boucle.

insérée dans un bloc conditionnel pour court-circuiter la fin de boucle

```
#!/bin/sh
# fichier continue.sh
for fic in *.sh
do
    echo "< fichier ${fic} >"
    if [ ! -r "${fic}" ]
    then
        echo "*****"
        echo "fichier ${fic} non lisible"
        continue # sauter la commande head
    fi
    head -4 ${fic}
done
exit 0
```

21.4.4 Redirections et boucles

redirection (d'entrée ou de sortie) après done

⇒ s'applique à la structure itérative

Exemple

```
#!/bin/sh
# redirection et structure itérative
# version à conseiller
for i in 1 2 3
do
    echo $i
done > resultat # redirection après done
exit 0
```

Éviter la méthode suivante :

```
#!/bin/sh
# redirection et structure itérative
# version à déconseiller
# partir d'un fichier vide
cat /dev/null > resultat
for i in 1 2 3
do
    echo $i >> resultat # accumuler dans la boucle
done
exit 0
```

22 Exemple commenté d'un script

22.1 Introduction

Comment passer tous les noms des fichiers d'un répertoire en minuscules ?
(le chemin du répertoire sera passé en argument de la commande)

Principales commandes utilisées :

- Changer le nom d'un fichier

```
mv FIC1.F90 fic1.f90
```

- Passer en minuscules (pas de signes diacritiques en UTF8)

```
tr '[:upper:]' '[:lower:]'
```

- Faire une boucle sur tous les fichiers du répertoire

```
for f in *
```

```
do
```

```
...
```

```
done
```

22.2 Le cœur de script

```
for NOM in *
do
    passer NOM en minuscules avec tr ⇒ nom
    mv ${NOM} ${nom}
done
```

Mais **tr** est un filtre qui transforme l'entrée standard, donc il faut afficher le nom initial sur la sortie standard par **echo**.

```
echo ${NOM} | tr '[:upper:]' '[:lower:]'
```

Puis récupérer la sortie standard de ce tube dans une variable **nom** grâce à la syntaxe **\$()**.

```
nom=$(echo ${NOM} | tr '[:upper:]' '[:lower:]')
```

22.3 Version minimale du script

```
#!/bin/sh
# fichier min-noms-0.sh
# passage en minuscules des noms des fichiers d'un répertoire
# version minimale
cd $1
for NOM in *
do
    # traduction du nom en minuscules
    nom=$(echo ${NOM} | tr '[:upper:]' '[:lower:]')
    # changement effectif du nom de fichier
    mv ${NOM} ${nom} && echo ${NOM} '=>' ${nom}
done
echo "fin"
exit
```


Problèmes :

— on peut écraser par exemple `fic1` en traitant `Fic1`

⇒ ne pas renommer dans ce cas ... sauf si `fic1` est vide.

⇒ vérifier si `mv` risque d'écrire sur un autre fichier déjà présent et non vide.

```
if [ "${nom}" != "${NOM}" ]
then # les noms diffèrent effectivement
    echo on va essayer de changer ${NOM} en ${nom}
    if [ -s "${nom}" ]
    then # risque d'écraser un fichier non vide
        echo ...
    else
        mv ${NOM} ${nom}
    fi
fi
```

— que faire s'il ne s'agit pas d'un fichier ordinaire ?

⇒ ne pas renommer dans ce cas (répertoire par exemple)

22.4 Version élémentaire du script

```
#!/bin/sh
# fichier min-noms-1.sh
# passage en minuscules des noms des fichiers d'un répertoire
# version élémentaire
cd $1
for NOM in *
do
    if [ -f "${NOM}" ]
    then
        # le fichier $NOM existe et est un fichier ordinaire
        # traduction du nom en minuscules
        nom=$(echo ${NOM} | tr '[:upper:]' '[:lower:]')
        if [ "${nom}" != "${NOM}" ]
        then
```

```
# les noms diffèrent effectivement
echo "peut-on changer ${NOM} en ${nom} ?"
if [ -s "${nom}" ]
then # risque d'écraser un fichier non vide
    echo "${NOM} devrait écraser ${nom} '=>' non traité >&2
else # changement effectif du nom de fichier
    mv ${NOM} ${nom} && echo "${NOM} '=>' ${nom}"
fi
fi
else
    # le fichier n'existe pas ou n'est pas un fichier ordinaire
    echo "${NOM} n'est pas un fichier ordinaire => non traité" >&2
fi
done
echo "fin"
exit
```

Problème plus grave : que se passe-t-il si on ne passe pas d'argument ?

`cd $1` ⇒ `cd` et on modifie les noms dans **le répertoire d'accueil** !

⇒ Vérifier s'il y a un argument,

sinon on peut **choisir de** travailler dans le répertoire courant.

S'il y a un argument, s'assurer que la commande `cd $1` réussit, ou plus précisément, arrêter le script si elle échoue.

```
if ! cd ${repertoire}
then
    echo "$repertoire inaccessible" >&2
    exit 2
fi
```

Ajouter quelques éléments de contrôle avec une liste avant et après les renommages, mais sans déposer de fichier temporaire dans le répertoire de travail.

22.5 Version plus robuste du script

```
#!/bin/sh
# fichier min-noms.sh
# passage en minuscules des noms des fichiers d'un répertoire
# test sur le nombre d'arguments
case $# in
  0) repertoire="."
    ;;
  1) repertoire=$1
    ;;
  *) echo erreur nombre d\arguments
     echo "usage: $0 [répertoire]" >&2
     exit 1
    ;;
esac
```

```
if [ ! -d "${repertoire}" ]
then
    echo "${repertoire} n'est pas un répertoire => abandon" >&2
    exit 2
fi
if ! cd ${repertoire}
then
    echo "répertoire inaccessible => abandon" >&2
    exit 2
fi
echo "passage en minuscules du nom des fichiers de $(pwd)"
echo 'Confirmez svp par O/N [N]'
OK=O
read reponse
if [ "${reponse}" != "${OK}" ]
then
```

```
    echo abandon demandé
    exit 0
fi
TEMPFILE="/tmp/$(whoami) .$$" # fichier temporaire de nom unique
ls -l > ${TEMPFILE}          # liste avant modifications
for NOM in *
do
    if [ -f "${NOM}" ]
    then
        # le fichier $NOM existe et est un fichier ordinaire
        # traduction du nom en minuscules
        nom=$(echo ${NOM} | tr '[:upper:]' '[:lower:]')
        if [ "${nom}" != "${NOM}" ]
        then # les noms diffèrent effectivement
            echo "peut-on changer ${NOM} en ${nom} ?"
            if [ -s "${nom}" ]
```

```
    then # risque d'écraser un fichier non vide
        echo ${NOM} devrait écraser ${nom} '=>' non traité >&2
    else # changement effectif du nom de fichier
        mv ${NOM} ${nom} && echo ${NOM} '=>' ${nom}
    fi
fi
else
    # le fichier n'existe pas ou n'est pas un fichier ordinaire
    echo "${NOM} n'est pas un fichier ordinaire => non traité" >&2
fi
done
ls -l > ${TEMPFILE}+ # liste après modifications
echo "Bilan"
diff ${TEMPFILE} ${TEMPFILE}+ # comparaison des listes
/bin/rm ${TEMPFILE} ${TEMPFILE}+ # ménage
exit 0
```


22.6 Limitations

- Fichiers cachés (commençant par « . ») non traités
 - ⇒ remplacer `for NOM in *` par
`for NOM in $(ls -a)` (`.` et `..` éliminés car répertoires)
- Pas d'action en cas de collision de noms,
mais on pourrait demander de saisir un autre nom
- On peut interrompre le déroulement en cours de boucle
 - ⇒ insérer `trap '...' INT` pour nettoyer les fichiers temporaires
- Cas des noms comportant des caractères spéciaux non étudié
- `tr` ne traite que les caractères sur un octet,
 - ⇒ aucun caractère accentué en UTF-8 dans les noms n'est traité

23 Compléments sur le shell

23.1 Commandes internes

Certaines commandes intégrées au shell (*builtin*) \Rightarrow plus rapides, ne lancent pas un nouveau processus, permettent d'affecter le shell courant...

cd, echo, pwd, read, set, ...

eval, exec, getopts, ...

Pas de `man`, sauf celui du shell, mais **help** *cmde_interne*

23.2 Exécution dans le shell courant

\Rightarrow hériter des variables _____ en sh, bash ou ksh _____

```
• ._commande
```

Exemple : `._profile`

23.3 Exécution avec les droits admin via `sudo`

```
sudo commande
```

permet d'exécuter la commande *commande* avec les droits de l'administrateur nécessite autorisation préalable pour les commandes concernées.

23.4 Autres commandes internes

23.4.1 La commande `eval`

Dans certaines circonstances, nécessité de faire agir le shell 2 fois sur la ligne de commande \Rightarrow double interprétation par le shell.

Cas le plus classique : accès au contenu du contenu d'une variable :

``${variable}`` \Rightarrow utiliser **eval**

eval valeur=`\`${variable}``

protéger le premier `$` de la première interprétation par le shell, sinon erreur de syntaxe

Exemple : affichage du dernier argument positionnel d'un script :

Si le script `test-eval.sh` contient :

```
i=$#
```

```
echo variable \${${i}}
```

```
eval echo valeur \${${i}}
```

L'appel `test-eval.sh un deux trois`

affichera le nom du dernier paramètre puis sa valeur, par exemple :

```
variable ${3}
```

```
valeur trois
```

23.4.2 La commande `exec`

exec *commande* vient remplacer le processus courant par celui de *commande*

Si `exec commande` est lancé en interactif, il y a fermeture du shell, donc de la session à la fin de la commande. \Rightarrow `exec csh` pour passer en `csh`.

`exec > fichier` (sans commande) en début de script

\Rightarrow redirection de sortie pendant tout le script.

23.5 Divers

23.5.1 Alias du shell

Notion d'alias scrutés **avant** les commandes

Choix d'options des commandes existantes, raccourcis pour des commandes,...

```
alias ls='ls -F' force l'option -F (Flag)
```

```
alias rm='rm -i' force l'option de confirmation
```

```
alias la='ls -a' pour voir les fichiers cachés
```

```
\ls permet de retrouver la commande ls native.
```

23.5.2 Identifier une commande **type**

type permet de savoir comment est interprété un identificateur,

avec l'ordre de priorité suivant :

alias, mot-clef, fonction, commande interne, shell-script ou exécutable

```
type ls          affiche          ls est un alias vers « ls -F »
```

23.5.3 Affichage d'une progression arithmétique seq

seq [*premier* [*incrément*]] *dernier*

affiche la progression arithmétique depuis *premier* (1 par défaut)

jusqu'à *dernier*, par pas de *incrément* (1 par défaut)

Option **-s** pour spécifier le **s**éparateur (retour ligne par défaut)

Option **-w** pour afficher chaque nombre avec autant de chiffres (*equal-width*)

seq -s' ' 10 2 15 affiche **10 12 14**

seq -s'/' 2 5 affiche **2/3/4/5**

seq -s'-' 5 affiche **1-2-3-4-5**

seq -s' ' 8 12 affiche **8 9 10 11 12**

seq -s' ' -w 8 12 affiche **08 09 10 11 12**

23.5.4 Récursivité

Un script peut s'appeler lui-même, tant que le nombre de processus lancés ne dépasse pas la limite fixée par l'administrateur (voir `ulimit`).

Méthode récursive élégante, mais souvent peu performante.

23.5.5 Fichiers d'initialisation du shell

En `ksh` et `bash` (différent avec `csh` et `tcsh`)

- `/etc/profile` pour tous au login
- `${HOME}/.profile` ou `${HOME}/.bash_profile`
personnels au login
- éventuellement le fichier défini par la variable `ENV`
`${HOME}/.kshrc` ou `${HOME}/.bashrc`

23.6 Automatisation des tâches avec la commande `make`

Outil de gestion des dépendances entre des fichiers sur la base de leur date de modification et de règles de dépendances.

Application la plus classique : reconstituer un programme exécutable à partir des fichiers sources en ne recompilant que ceux qui ont été modifiés.

- **cible** (*target*) : fichier à produire
- **règle** de production (*rule*) : liste des commandes à exécuter pour produire une cible (compilation pour les fichiers objets, édition de lien pour l'exécutable)
- **dépendances** : ensemble des fichiers nécessaires à la production d'une cible

Le fichier **makefile** liste les cibles, décrit les dépendances et les règles.

Il est construit à partir de l'arbre des dépendances.

make *cible*

lance la production de la *cible* en exploitant le fichier `makefile`