Utiliser le système UNIX

Département d'Informatique
Faculté des Sciences de Luminy

Henri Garreta, Jean-luc Massat
DESS CCI - Janvier 1996 (révision 2001)

Table des matières

Chapitre 1

Système de fichiers

En règle générale, tout fichier est vu par UNIX comme une entité capable de consommer, de conserver ou de fournir des *suites homogènes d'octets*. Le système n'impose aucune structure particulière aux fichiers et, sauf pour certains fichiers qui le concernent directement (les fichiers spéciaux et les répertoires, voir ci-dessous), il ne fait aucune supposition quant à leur contenu, cela ne regarde que les programmes qui construisent ou qui exploitent les fichiers.

Chaque fichier est décrit par un enregistrement, appelé *i-node*, qui en précise les principales caractéristiques : son type, sa taille, la date de sa création et celle de sa plus récente modification, le numéro du propriétaire du fichier et de son groupe, les permissions, le nombre de liens sur le fichier, etc. Les *i-node* sont rangés dans une table et chacun est identifié par un numéro, son rang dans cette table. Malgré leur rôle fondamental, les i-nodes n'interviennent pratiquement jamais dans l'utilisation courante du système. UNIX distingue trois sortes de fichiers :

- Un *répertoire*, *catalogue* ou encore *directory*, est un fichier dont le contenu est une liste de couples (*nom*, *indice d'un i-node*). Chaque nom est donc un renvoi à un fichier. Parmi ces fichiers, certains sont à leur tour des répertoires, et contiennent d'autres répertoires, etc. Ainsi, les répertoires définissent une structure d'arbre sur l'ensemble des noms des fichiers. Dans un système qui n'est pas endommagé, tout fichier est accessible par au moins un nom appartenant à un répertoire, sauf un fichier particulier, un répertoire qu'on appelle la racine. La racine du système de fichiers n'a pas de nom car, pour un fichier, « avoir un nom » est la même chose que « figurer dans un répertoire ».

- Les *fichiers spéciaux*. Les unités matérielles d'entrée/sortie figurent également dans le système de fichiers, parmi les « vrais » fichiers sur support magnétique. Cela permet l'écriture de programmes homogènes, indépendants des particularités techniques des opérations d'entrée/sortie effectuées. Ainsi, du point de vue du programmeur, la procédure pour lire ou écrire dans un fichier ordinaire est la même que celle qui permet d'échanger de l'information avec un appareillage quelconque.

- Les *fichiers ordinaires* sont tous les autres fichiers, c'est à dire les fichiers sur disque magnétique qui ne sont pas connus comme étant des répertoires. Pour l'utilisateur, deux distinctions principales, que le système ignore : les *fichiers de texte* sont imprimables, éditables, affichables, etc., alors que les fichiers binaires ne le sont pas. De texte ou binaires, les *fichiers exécutables* représentent des programmes et sont susceptibles de prendre le contrôle de machine, tandis que les fichiers non exécutables ne représentent que des données.

1.1 Noms de fichier

Références absolues. La manière de nommer les fichiers en UNIX utilise la structure arborescente des répertoires. Tout nom de fichier figure dans un répertoire qui, s'il n'est pas la racine, figure à son tour dans un autre répertoire lequel, s'il n'est pas la racine, figure lui-même dans un répertoire, etc. On fait référence à un fichier particulier en écrivant tous ces noms de répertoires, séparés par / :

```
/usr/local/bin/jeux/morpion
```

Cette expression désigne le fichier « `morpion` » du répertoire « `jeux` » du répertoire « `bin` » etc. On dit que « `/usr/local/bin/jeux/` » est un *chemin absolu* qui mène à notre fichier à partir de la racine. Un chemin spécifie un répertoire ; en principe il peut indifféremment se terminer ou non par un « / ». Ainsi, « / » tout seul est une référence absolue à la racine du système de fichiers.

Répertoire de travail et références relatives. Pour alléger les références aux fichiers, un répertoire particulier est constamment distingué comme étant le *répertoire de travail*. Un chemin qui ne commence pas par / est alors considéré comme un *chemin relatif* : son premier élément doit être recherché dans le répertoire de travail, non dans la racine. Par exemple, si le répertoire de travail est « `/usr/local` », alors « `bin/jeux/morpion` » est une référence relative au fichier « `/usr/local/bin/jeux/morpion` ».

Auto-références. Chaque répertoire r contient deux noms spéciaux, au rôle prédéfini : « `.` », qui renvoie au répertoire r lui-même, et « `..` » qui renvoie au répertoire ancêtre de r. Par conséquent, le chemin « `.` » est une référence relative au répertoire de travail. Par exemple, si le répertoire de travail est « `/usr/local` » alors « `.` » est la même chose que « `/usr/local` » et « `./bin/jeux/morpion` » est encore une référence relative au fichier « `/usr/local/bin/jeux/morpion` ».

De la même manière, le chemin double « `..` » est une référence relative au répertoire *ancêtre* du répertoire de travail. Par exemple, si le répertoire de travail est « `/usr/local` » alors « `..` » signifie « `/usr` » et « `../bin/jeux/morpion` » est une référence relative au fichier « `/usr/bin/jeux/morpion` ».

Il en découle enfin, mais cela n'a aucune utilité, que des chemins comme « `/usr/local/./bin` », « `/usr/local/./././.bin` » ou « `/usr/local/../local/../local/bin` » sont équivalents à « `/usr/local/bin` ».

1.2 Permissions

Chaque utilisateur d'un système UNIX possède un *numéro d'utilisateur* unique, et un *numéro de groupe* qui est commun à plusieurs utilisateurs. L'un et l'autre sont attribués une fois pour toutes par le responsable du système, lors de l'enregistrement initial de l'utilisateur. Un groupe est une collection arbitraire d'utilisateurs, par exemple tous ceux qui travaillent sur un même projet.

Les *permissions des fichiers* sont définies relativement à trois domaines : *le propriétaire* du fichier, qui est généralement son créateur, *le groupe*, c'est à dire les utilisateurs ayant le même numéro de groupe que le propriétaire du fichier, et *les autres* utilisateurs du système. Pour chacun de ces trois domaines, trois opérations peuvent être séparément permises ou interdites : la lecture, l'écriture et l'exécution. On représente ces permissions par trois groupes de trois bits (1 pour permis, 0 pour interdit), dans l'ordre

$$(\mathit{read,write,exec})_{owner}\ (\mathit{read,write,exec})_{group}\ (\mathit{read,write,exec})_{other}$$

ou bien par trois groupes de trois caractères, `r`, `w` ou `x` pour 1, `-` pour 0. Par exemple, « `rwxr-x--x` » indique un fichier sur lequel le propriétaire a tous les droits, les membres de son groupe ceux de le lire et de l'exécuter et les autres utilisateurs uniquement celui de l'exécuter.

Si le fichier est un répertoire, l'*exécuter* veut dire le faire figurer dans un chemin ou en faire le répertoire de travail, tandis que y écrire signifie lui ajouter ou lui enlever des fichiers. On retiendra que la permission de créer ou de supprimer un fichier équivaut à la permission d'écrire sur le répertoire auquel le fichier appartient.

Chapitre 2

Le shell, début

Le shell est l'interprète des commandes d'UNIX. C'est le programme qui dialogue avec vous aussi longtemps que votre poste de travail n'est pas sous le contrôle d'une commande spécifique. Dès que la procédure de connexion (*login*) est terminée, le shell prend le contrôle et vous écoute. Il lit chaque commande que vous tapez, il en développe les éléments génériques (comme les familles de noms de fichiers) et il y remplace les éléments symboliques (comme les variables d'environnement). De plus, le shell comporte un mécanisme d'enchaînement des commandes qui en fait un vrai langage de programmation.

A cause de son rôle d'interface, le shell représente pour l'utilisateur la « face visible » d'UNIX. On pourrait croire qu'il s'agit d'un programme très spécial, aux privilèges exorbitants. Il n'en est rien. Mis à part le fait d'être automatiquement lancé par le système au début d'une session[1], le shell est un programme comme les autres, sans aucun droit particulier. Plusieurs interprètes de commandes sont disponibles, parfois sur une même machine ; de plus, rien ne vous empêche d'écrire votre propre shell. Les interprètes de commandes les plus répandus sont le « *Bourne shell* » (`sh`), du nom de son auteur Steve Bourne, et le « *C shell* » (`csh`), ainsi appelé parce que, comme langage de programmation, il est assez proche de C. Le *C shell* est plus riche et plus régulier que le *Bourne shell*, mais nous étudierons ce dernier car il est plus répandu et plus simple.

Pour chaque utilisateur, le nom du shell qui doit être activé lors de sa connexion est précisé dans le fichier `/etc/passwd` (c'est le dernier champ de la ligne le concernant). C'est le responsable du système qui définit ou modifie ce genre d'informations. Si les machines sont organisées en réseau, il est possible que le fichier `/etc/passwd` de chaque machine ne reflète pas l'ensemble des utilisateurs et des mots de passe.

2.1 Début et fin de la session

Si sur un poste donné aucun utilisateur n'est connecté au système, alors l'écran doit afficher la question suivante (si ce n'est pas le cas, appuyez sur la touche retour-chariot, elle apparaîtra) :

```
pcN login:
```

en réponse vous devez taper le « nom d'utilisateur » que vous a attribué le responsable du système. Attention, vous devez taper ce nom en minuscules, autrement UNIX croira que vous utilisez un mauvais terminal ne disposant que des majuscules et la suite deviendra pénible.

Si vous avez défini un mot de passe, la question suivante est :

```
Password:
```

vous devez y répondre en tapant votre mot de passe secret. Exemple de dialogue (les interventions de l'utilisateur sont soulignées) :

[1] Un shell lancé par UNIX lors de la connexion s'appelle un login shell. Il reste actif tant que l'utilisateur ne se déconnecte pas. Au cours de la session, l'utilisateur peut lancer d'autres shell, qui s'exécutent alors comme des programmes ordinaires.

```
UNIX(r) System V Release 4.0 (big)
login: zigomar
Password: ~~~~~~
Login incorrect   (password erroné : connexion ratée)
login: zigomar
Password: ~~~~~~
Last login: Mon Nov 24 08:52:39 from pc30
You have mail.
big:~$
```

Si la connexion réussit, le système affiche quelques informations d'intérêt général, comme la date de votre dernière connexion, le fait que vous avez reçu du courrier que vous n'avez pas encore lu, ou des messages de la part du responsable du système, puis active un shell qui se met en attente de vos commandes.

Il est fortement conseillé, pour ne pas dire obligatoire, surtout s'il s'agit d'une machine connectée à un réseau informatique, d'installer un mot de passe, pour se mettre à l'abri de la malveillance et, surtout, de l'étourderie des autres utilisateurs. Pour définir ou changer votre mot de passe, composez la commande

```
passwd
```

Elle vous demandera de composer (en aveugle, tout cela est secret) l'ancien et le nouveau mot de passe, ce dernier deux fois. Sur les machines que nous utilisons (c'est-à-dire des PC équipés de Linux) la commande `passwd` est remplacée par `yppasswd`.

FIN DE SESSION. Pour terminer une session (du *Bourne shell*) vous devez taper une ligne réduite à l'unique caractère *Ctrl-D* (maintenez la touche *Ctrl* pressée pendant que vous appuyez sur la touche *D*). Attention, si la ligne ne *commence* pas par *Ctrl-D* cela ne marchera pas, vous devrez la refaire. Une session du *C shell* se termine généralement en composant la commande

```
logout
```

2.2 Principe général des commandes

Dès la fin de la procédure de connexion, chaque fois qu'une commande particulière ne détient pas le contrôle, le *shell* est à votre l'écoute. Il manifeste ce fait par l'affichage d'un *prompt*, généralement (en Bourne shell) le caractère $. Vous devez alors composer une ligne, qui ne sera prise en compte que jusqu'à la frappe d'un retour-chariot. Le plus souvent *une* ligne contient *une* commande, mais ce n'est pas obligatoire, car on peut enchaîner plusieurs commandes sur une même ligne, et certaines commandes s'étendent sur plusieurs lignes.

Le premier mot du texte que vous composez pour activer une commande identifie la commande en question, c'est son nom. Il est généralement suivi de plusieurs autres expressions qui indiquent des options et des paramètres de la commande. Il y a trois sortes de commandes, qu'on utilise de la même manière : les commandes internes, les programmes et les *scripts*.

Les commandes internes expriment les fonctionnalités du *shell* lui-même ; elles sont reconnues et exécutées sans l'aide d'un autre fichier. Toute commande qui n'est pas interne est supposée être le nom d'un fichier exécutable, qui peut être soit un fichier du système, c'est à dire qui a été livré avec UNIX, soit un fichier crée par un utilisateur. Dans un cas comme dans l'autre ce fichier peut être un fichier binaire exécutable, par exemple le résultat de la compilation d'un programme écrit en C, ou bien une procédure de commande ou *script*. Un *script* est un fichier de texte entièrement composé de commandes.

Comme les commandes externes n'appartiennent presque jamais au répertoire de travail, il serait très pénible de devoir les spécifier par les références complètes des fichiers correspondants. Pour l'éviter, le *shell* maintient constamment une liste de répertoires où chercher les commandes, c'est la valeur de la variable d'environnement[2] `PATH`. Ainsi, si la valeur de cette variable est la chaîne « . :/bin :/usr/bin » et si vous tapez le mot « job » (qui n'est pas le nom d'une commande interne) alors le *shell* tentera d'exécuter « ./job » puis, en cas d'échec, « /bin/job », puis encore « /usr/bin/job ».

[2]Nous expliquerons plus loin comment on donne une valeur à une « variable d'environnement ».

2.3 Options et paramètres

La quasi totalité des commandes prédéfinies (et vos programmes aussi, si ils ont l'esprit UNIX) suivent les conventions suivantes :

- Le nom de la commande est souvent très court, donc rarement expressif
- Après le nom de la commande on trouve un nombre quelconque d'options, de la forme

  `-`*`caractère`*

  dans le cas d'une option qui n'exige pas une valeur, et

  `-`*`caractère valeur`*

  lorsqu'une valeur est nécessaire. La nature de l'option, indiquée par le caractère qui lui sert de nom, détermine si une valeur doit ou non être présente.
- Dans beaucoup de commandes, si aucune option ne comporte de valeur ou si une seule option en comporte, on doit regrouper les options. Par exemple on nous fait écrire « `-abc` » à la place de « `-a -b -c` » et « `-xvf` *`valeur`* » à la place de « `-x -f` *`valeur`* `-v` ».
- L'ordre des options entre elles est sans importance. L'ordre des noms de fichier entre eux dépend de la nature de la commande (voir le point suivant)
- Après le nom de la commande on trouve un nombre quelconque de noms de fichiers, qui sont les *entrées* de la commande. Selon la nature de cette dernière, le travail sera fait soit sur la concaténation des fichiers, soit successivement sur chacun d'eux
- Lorsqu'aucun nom de fichier n'a été indiqué, la commande lit ses données éventuelles sur son entrée standard. Chaque fois que la commande le permet, les données sont sous forme textuelle.
- La commande écrit ses résultats éventuels sur sa sortie standard. Chaque fois que la commande le permet, ces résultats sont sous forme textuelle. Ces textes sont aussi laconiques que possible.

Exemples :

```
cat fic1 fic2 fic3
```

la commande `cat` (pour *catenate*) est tout à fait minimale : elle ne fait qu'écrire les caractères qu'elle lit. Mais elle suit les conventions indiquées. Par conséquent, l'expression ci-dessus affiche la concaténation des trois fichiers fic1, fic2 et fic3.

```
cc -g -o monfic.o -c monfic.c
```

cette expression est un appel du compilateur C (cc, pour *C compiler*) avec les options « -g », « -o monfic.c » et « -c » (les options -g et -c de la commande cc ne requièrent pas de valeur, l'option -o oui), et avec pour entrée le fichier « monfic.c ». Cette commande n'écrit rien sur la sortie standard.

2.4 Noms de fichier génériques.

Beaucoup de commandes du *shell* ont des noms de fichier pour arguments. Ces noms peuvent comporter les spécifications génériques `*`, `?` et `[...]` avec la signification suivante :

`*`	n'importe quelle chaîne de caractères (y compris la chaîne vide),
`?`	n'importe quel caractère unique,

`[...]` n'importe quel caractère unique parmi ceux d'un ensemble. On a droit aux spécifications :

- c le caractère c,
- c_1-c_2 les caractères compris entre c_1 et c_2.

Ces expressions sont comprises comme des motifs de recherche dans l'ensemble des noms de fichiers *réellement existants* dans le système. A l'endroit où elle figure, l'expression générique représente la liste des noms de fichier qui s'unifient avec le motif de recherche. Par exemple, l'expression

```
*.c
```

représente la liste de tous[3] les fichiers du répertoire de travail dont le nom se termine par « .c », l'expression

```
/tmp/*.[ch0-9]
```

représente la liste des fichiers du répertoire `/tmp` dont le nom se termine par un point suivi de c, h ou un chiffre, et l'expression

```
/usr/users/henri/*/core
```

représente la liste de tous les fichiers de nom core qui se trouvent dans un sous-répertoire immédiat (un niveau) de `/usr/users/henri` (comme `/usr/users/henri/p2c/core`).

N.B. Afin que le mécanisme des noms de fichier génériques puisse être utilisé de manière tout à fait générale, il faut qu'à tout endroit où peut figurer un nom de fichier puisse figurer aussi bien une liste de noms de fichier. C'est le cas de la plupart des commandes du système, et il est conseillé d'écrire ses propres commandes dans le même esprit. Selon la nature de la commande, le traitement d'une liste de noms de fichiers se traduira soit par la concaténation de ces fichiers, comme

```
$ sort liste[0-9].txt > liste_generale.txt
```

(concaténation et tri des fichiers `liste0.txt`, `liste1.txt`, etc.), soit par la répétition de la commande sur chaque élément de la liste, comme

```
$ cc -c module*.c
```

(compilation de `moduleA.c`, puis `moduleB.c`, etc.)

2.5 Déroutement des E/S et tubes.

L'entrée et la sortie standards ne sont pas fixées lors de l'écriture de la commande. En l'absence d'une indication contraire, pendant une session interactive l'entrée est le clavier du poste de travail, et la sortie est l'écran de ce même poste. Cependant, la spécification

`< `*fichier*

déroute l'entrée standard : le programme ne lit plus le clavier mais le fichier indiqué, qui doit être un fichier de texte. Cette substitution est complètement transparente, aucune précaution particulière n'était à prendre lors de l'écriture du programme correspondant à la commande. En particulier, lors de la rencontre de la fin du fichier la commande ressent exactement la même chose que lorsque l'utilisateur tape *Ctrl-D* au clavier. De la même manière, la spécification

`> `*fichier*

[3] Sauf les fichiers dont le nom commence par « . », ce qui est en général souhaitable.

déroute la sortie standard : les écritures du programme ne se font pas à l'écran, mais dans un fichier nouvellement crée ayant le nom indiqué. Attention : tout fichier de même nom est préalablement détruit. La spécification

```
>> fichier
```

ne diffère de la précédente que par le point suivant : si *fichier* existe déjà, alors il n'est pas écrasé, les nouvelles écritures se font en fin du fichier, à la suite des précédentes.

Exemple (la commande `ls` liste les fichiers du répertoire courant, la commande `wc -l` compte le nombre de lignes de l'entrée standard) :

```
$ ls > listefics
$ wc -l < listefics
32
$
```

Bien entendu, on peut dérouter simultanément l'entrée et la sortie d'une commande. Exemple (la commande sort écrit sur la sortie standard les lignes lues sur l'entrée standard, triées par ordre alphabétique) :

```
$ sort < fic_brut > fic_trie
```

Un *tube* (*pipe*) réalise aussi un déroutement des E/S, en même temps qu'il enchaîne l'exécution de deux commandes. La spécification

```
commande1 | commande2
```

déclenche l'exécution *simultanée* des deux commandes indiquées, après que la sortie de *commande1* ait été déroutée sur l'entrée de *commande2*. On dit que l'on a relié ces deux commandes par un tube. Elles s'exécutent en parallèle, mais elles sont de fait synchronisées par le tube : si *commande1* écrit plus vite que *commande2* ne lit, le tube se remplira et *commande1* se trouvera bloquée jusqu'à ce que *commande2* en lisant y ait libéré de la place. De la même manière, si *commande2* lit plus vite que *commande1* n'écrit, le tube se videra et c'est *commande2* qui sera bloquée jusqu'à ce que *commande1* ait fait de nouvelles écritures.

Exemple : nous avons donné ci-dessus l'exemple « `ls > listefics` » immédiatement suivi de « `wc -l < listefics` ». Il est clair que si notre seul but était de compter les éléments du répertoire courant, alors le fichier listefics n'a aucun intérêt en lui-même et notre commande gagnera à être écrite sous la forme :

```
$ ls | wc -l
32
$
```

Du point de vue des E/S, deux commandes reliées par un tube comme « *commande1* | *commande2* » constituent un tout possédant une unique entrée, celle de *commande1*, et une unique sortie, celle de *commande2*. Ces deux unités peuvent être déroutées à leur tour, ou servir à monter un autre tube. Nous en verrons des exemples dans la suite.

Chapitre 3

Principales commandes

3.1 Commandes générales

Dans les descriptions suivantes, la construction [*expr*] indique que *expr* est optionnelle, et la construction *expr ... expr* indique une suite d'un nombre quelconque de *expr* (peut être aucune).

echo *argument* **...** *argument*

Affiche les *arguments* sur sa sortie standard. Elle ne lit pas d'entrée.

```
$ echo Bonjour a tous
Bonjour a tous
$
```

Dans les scripts, cette commande joue le rôle du *writeln* de Pascal. L'option `-n` supprime l'ajout d'un retour-chariot en fin de ligne (*writeln* devient *write*). En dehors d'un script, cette commande est bien utile pour visualiser la valeur d'une variable ou d'un nom générique avant de les donner pour argument à une autre commande :

```
$ echo $HOME
/usr/users/henri
$
```

cat *fichier* **...** *fichier*

(*Catenate*) Écrit sur la sortie standard la concaténation des fichiers indiqués ou, si aucun fichier n'est indiqué, l'entrée standard. Les principales utilisations de cette commande sont :

- L'affichage d'un ou plusieurs fichiers :

```
$ cat monfic.c
ici apparaîtront les lignes de monfic.c
```

- La concaténation de fichiers :

```
$ cat fic1.c fic2.c fic3.c > fic-tout.c
$
```

- La création manuelle d'un fichier (petit, sinon il vaut mieux utiliser un éditeur de texte !)

```
$ cat > fic-nouveau.c
ici on tape les lignes qui formeront fic-nouveau.c
Ctrl-D
$
```

more *fichier ... fichier*

Affiche le(s) fichier(s) indiqué(s), ou l'entrée standard, page par page selon la taille de l'écran du poste de travail. L'utilisateur peut taper :

- un blanc, pour produire l'affichage d'une page supplémentaire
- un retour-chariot, pour produire l'affichage d'une ligne supplémentaire
- le caractère « q » pour quitter ce programme

who

Affiche les noms des utilisateurs présentement connectés au système.

mail *destinataire ... destinataire*

Employée avec un ou plusieurs arguments, cette commande copie le contenu de l'entrée standard dans les boîtes aux lettres des utilisateurs indiqués. Employée sans argument, elle vous permet de lire le courrier qui se trouve dans votre boîte aux lettres. Pour commencer, le plus récent message est affiché. Vous pouvez ensuite frapper une des commandes suivantes :

Retour-chariot	passer au message suivant
`d`	détruire ce message et passer au suivant
`p`	réafficher le message courant
`-`	revenir au message précédent
`s` *fichier*	écrire le message courant dans le fichier indiqué
`m` *user ...*	envoyer le message courant aux utilisateurs indiqués
`q`	quitter ce programme

Les *destinataires* peuvent être indiqués sous deux formes principales

`username`	spécifie un utilisateur de votre machine (ou de votre grappe de machines, réseau local, etc.)
`username@`*machine*	spécifie un utilisateur connu sur la machine (appartenant à un réseau auquel vous êtes connectés) dont l'adresse est indiquée. Cette adresse doit suivre la syntaxe du réseau en question.

Exemple (adresses *Internet*, le grand réseau mondial des utilisateurs d'UNIX) :

```
$ mail basingerk@mgm.hollywood-ca.us
$ mail massat@lim.univ-mrs.fr
```

write *username*

Cette commande permet d'envoyer un message à un autre utilisateur actuellement connecté sur *la même machine*. Exemple : Pierre ayant constaté la présence de Paul, il compose

```
$ write paul
Le rendez-vous de ce soir est annule
Ctrl-D
$
```

Paul voit alors apparaître sur son terminal un message de la forme

```
Message from pierre on tty12 at 19:40
Le rendez-vous de ce soir est annule
```

Cela établit une forme de communication à sens unique et immédiate, donc peut-être intempestive. De plus, le travail que Paul était en train d'exécuter sur son terminal a pu être très perturbé par l'affichage de ces textes. Pour cette raison, un utilisateur peut interdire la réception de tels messages par la commande

```
$ mesg n
```

On peut redevenir « réceptif » avec la commande `mesg y`.

3.2 Manipulation de fichiers

ls [*options* **]** *nom* **...** *nom*

(*List*) Affiche les contenus des répertoires et les informations sur les fichiers indiqués. Si aucun nom n'est indiqué, c'est le répertoire de travail qui est listé. Des options courantes sont :

`-l` format long. On obtient dans l'ordre : les permissions, le nombre de liens, le propriétaire, la taille en octets, la date de la dernière modification et finalement le nom du fichier (dans le format court on n'a que le nom). Exemple :

```
$ ls -l
drwxr-xr-x   2   jluc   1024   Feb 20 16:44   sources
-rwxr-x--x   1   jluc   3528   Feb 18  9:18   mcarr.c
...
```

Le premier caractère des permissions est « - » pour un fichier ordinaire, « d » pour un répertoire et un autre caractère pour un fichier spécial.

`-a` afficher tous les fichier du répertoire, y compris ceux dont le nom commence par « . », qui sont habituellement ignorés.

`-i` afficher le numéro de *i-node* en début de ligne pour chaque fichier

cd [*chemin* **]**

(*Change directory*) Définition du répertoire de travail. Si *chemin* est omis, la valeur de la variable `$HOME` est utilisée.

pwd

(*Print working directory*) Ecrit sur sa sortie standard le nom du répertoire de travail

cp *fichier1 fichier2*

cp *fichier* **...** *fichier répertoire*

(*Copy*) Copie de fichier. Dans la première forme, *fichier1* est copié sur *fichier2* (si *fichier2* existait déjà, il est écrasé). Dans la deuxième forme, les fichiers indiqués sont copiés, chacun conservant son nom, dans le répertoire indiqué. Dans un cas comme dans l'autre, cp refuse de copier un fichier sur lui-même.

mv *fichier1 fichier2*

mv *fichier ... fichier répertoire*

(*Move*) Changement du nom d'un ou plusieurs fichiers. Dans la première forme, le nom du fichier *fichier1* est changé en *fichier2*. Si *fichier2* existait déjà, tout se passe comme si on avait fait au préalable « `rm` *fichier2* » (voir page 13). Dans la deuxième forme, le renommage est tel que les fichiers disparaissent de leurs répertoires d'origine pour se retrouver, chacun avec son nom, dans le répertoire indiqué.

La commande `mv` ne joue que sur les *noms* des fichiers, en aucun cas elle ne recopie leur contenu. Par conséquent, on ne peut pas utiliser cette commande pour déplacer un fichier d'un volume dans un autre[1]. Voir la remarque sur les liens dans l'explication de `rm`, ci-dessous.

ln *fichier1 ... fichier2*

(*Link*) Création d'un lien sur un fichier. Il doit exister un fichier nommé *fichier1*. Alors, *fichier2* devient un autre nom pour ce même fichier, et on a une sorte de duplication de *fichier1* qui ne prend pas d'espace supplémentaire. Attention : si le contenu du fichier n'est pas stable, on peut avoir des problèmes de compréhension (les modifications de *fichier1* se répercuteront « magiquement » dans *fichier2*).

Dans le *i-node* du fichier concerné, le nombre de liens est augmenté de 1. Il n'y a pas de renvoi d'un fichier vers ses noms, et il n'y a pas de hiérarchie entre les noms d'un même fichier, ils sont tous aussi importants. D'autre part, on ne peut pas créer un lien vers un fichier qui appartient à un volume qui n'est pas celui auquel appartient le répertoire qui contient le nom en question. Voir la remarque sur les liens dans l'explication de `rm`, ci-dessous.

rm *fichier ... fichier*

(*Remove*) Au sens courant, impropre, cette commande supprime le fichier indiqué. En réalité elle ne supprime que le lien qui existe entre le nom *fichier* et le fichier ainsi nommé. Dans le *i-node* correspondant le nombre de liens est diminué de 1 ; si cela lui donne la valeur zéro alors, et seulement alors, le fichier sera supprimé. L'option `-r` permet de détruire un répertoire et son contenu (voir `rmdir` page 14).

Fichiers, noms de fichiers et liens

L'explication des commandes `mv`, `rm` et `ln` complète notre description du système des fichiers d'UNIX. En résumé : il y a d'une part l'ensemble des fichiers, décrits par les *i-node*. Ceux-ci sont rangés dans un tableau, il n'y a pas de hiérarchie parmi eux. D'autre part, certains fichiers, les répertoires, sont consacrés à la représentation d'un *arbre de noms*. Chaque nom renvoie à un unique fichier, on dit que le nom est un *lien* sur ce fichier. Plusieurs noms peuvent être liés au même fichier. Un fichier (i.e. un *i-node*) ne connaît pas ses noms, uniquement leur nombre. Quelque soit la circonstance dans laquelle cela se produit, quand le nombre de noms d'un fichier devient nul l'espace occupé par le fichier est libéré.

Ainsi, si nous notons f le fichier (ou le *i-node*, c'est la même chose) dont *fic1* est le nom :

`mv` *fic1 fic2*	supprime le lien *fic1* $\to f$ et crée un lien *fic2* $\to f$. S'il existait un fichier g de nom *fic2*, alors le lien *fic2* $\to g$ est détruit (cela peut entraîner ou non la destruction de g)
`ln` *fic1 fic2*	crée un lien *fic2* $\to f$ (sans détruire le lien *fic1* $\to f$). S'il existait un fichier g de nom *fic2*, alors le lien *fic2* $\to g$ est détruit
`rm` *fic1*	supprime le lien *fic1* $\to f$. Cela peut entraîner ou non la destruction de f)

[1] Une fois montés, les différents volumes apparaissent dans le système de fichiers comme des sous-arbres entiers. Rien ne distingue tel ou tel répertoire comme étant la racine d'un sous-arbre qui occupe un volume distinct de celui où se trouve la racine générale.

touch *fichier ... fichier*

La date de dernière modification des fichiers indiqués est changée comme si les fichiers venaient d'être modifiés. Voir la commande `make` pour un exemple d'utilisation d'une telle opération (page 31).

mkdir *chemin ... chemin*

(*Make directory*) Création des répertoires indiqués.

rmdir *chemin ... chemin*

(*Remove directory*) Effacement des répertoires indiqués. Une erreur est signalée si un de ces répertoires n'est pas entièrement vide.

chmod *mode fichier ... fichier*

(*Change mode*) Cette commande modifie les permissions des fichiers indiqués. On peut indiquer le mode de deux manières : un mode absolu, qui est un nombre entier dont l'écriture octale[2] est facile à obtenir à partir des permissions envisagées. Un mode relatif s'obtient en faisant suivre :

- une combinaison quelconque des lettres `u` (le propriétaire), `g` (son groupe) et `o` (les autres utilisateurs). L'absence de ce paramètre équivaut à « tous », c'est à dire « `ugo` ».
- un des signes `+` (pour ajoutez ce droit), `-` (pour enlevez ce droit) ou `=` (pour donnez ce droit uniquement)
- une combinaison quelconque des lettres `r` (lecture), `w` (écriture), `x` (exécution) ou `s`.

Exemple : la commande

```
chmod +x fichier
```

rend le fichier indiqué exécutable par tous les utilisateurs.

find *chemin ... chemin critère*

Cette commande parcourt les arborescences de fichiers ayant pour racines les chemins indiqués, à la recherche des fichiers vérifiant le critère indiqué. Ce critère est exprimé par la *conjonction* (« et ») d'expressions comme :

`-name` *nom* le fichier en cours d'examen a le *nom* indiqué. Vous pouvez utiliser dans *nom* des caractères génériques (voir page 7).

`-user` *nom* le fichier en cours d'examen appartient à l'utilisateur ayant le *nom* indiqué

`-group` *groupe* le fichier en cours d'examen appartient à un utilisateur du groupe indiqué (*groupe* est un nom ou un numéro)

`-atime` *nombre* le fichier en cours d'examen a été utilisé dans les *nombre* derniers jours

`-mtime` *nombre* le fichier en cours d'examen a été modifié dans les *nombre* derniers jours

`-newer` *fichier* le fichier en cours d'examen a été modifié plus récemment que le *fichier* indiqué

`-print` toujours vraie. Ce critère produit comme effet de bord l'écriture du nom du fichier en cours d'étude. Elle doit apparaître en dernière place dans la conjonction ; si elle est évaluée, c'est que les autres conditions ont été satisfaites, il y a donc lieu d'afficher le nom de ce fichier.

[2] Rappelons qu'un chiffre octal écrit en binaire prend trois bits. Par conséquent, un nombre octal (compris entre 000 et 777) est fait de trois paquets de trois bits, et la correspondance avec la représentation des permissions est évidente.

! *expr* Cette formule indique la condition contraire de celle qu'indique *expr*.

expr1 -o *expr2* La formule indique la disjonction (« ou ») à la place de la conjonction des conditions exprimées par *expr1* et *expr2*.

Exemple :

```
$ find /usr/users/henri -name moincarr.c -print
/usr/users/henri/UVC90/problemes/sources/moincarr.c
/usr/users/henri/UVC90/problemes/vieux_sources/moincarr.c
$
```

df *volume* **...** *volume*

(*Disk free*) Affiche le nombre de blocs disponibles sur le volume indiqué, ou sur tous les volumes. L'espace occupé ou disponible est donné aussi sous la forme de pourcentage de l'espace total.

tar [*options* **]** *fichier_ou_répertoire* **...** *fichier_ou_répertoire*

(*Tape archiver*) Cette commande est surtout utilisée pour piloter les transferts entre les fichiers ordinaires (sur disque magnétique) et les bandes magnétiques. Les bandes accomplissent deux fonctions essentielles : la sauvegarde des disques magnétiques et la distribution du logiciel. Exemples :

1. Sauvegarde de tout un sous-arbre du système de fichier sur une disquette au format UNIX :

   ```
   tar -cvf /dev/fd0 /usr/users/dessdc
   ```

2. Restauration de tout le contenu d'une cassette dans le répertoire courant :

   ```
   tar -xvf /dev/fd0
   ```

3. Savoir ce qu'il y a sur une disquette :

   ```
   tar -tvf /dev/fd0
   ```

Les options possibles sont :

`-x` extraction des fichiers à partir d'une archive,

`-t` listage des fichiers composants une archive,

`-c` construction d'une archive à partir des fichiers et des répertoires passés en paramètre.

`-v` fonctionnement verbeux (la commande rend compte des fichiers rencontrés au fur et à mesure)

`-f` *fichier* spécifier l'archive à utiliser ou à construire.

L'utilisation d'un périphérique d'archivage (*streamer*, *disquette*, *disque*) doit passer par la connaissance du fichier spécial associé à ce périphérique.

Attention : l'option `-c` implique l'effacement préalable de l'archive ; par conséquent, taper `-c` pour `-x` peut avoir un résultat catastrophique. **Conseil** : si le contenu d'une bande (ou disquette) est important, protégez physiquement celle-ci.

3.3 Utilitaires

wc [*options* **]** *fichier* **...** *fichier*

(*Word count*) Ecrit sur sa sortie standard le nombre de lignes et/ou de mots et/ou de caractères de la concaténation des fichiers indiqués. Les options sont :

`-l` compter les lignes

`-w` compter les mots

`-c` compter les caractères

L'option par défaut est `-lwc`. Exemple : « combien d'utilisateurs sont actuellement connectés ? »

```
$ who | wc -l
8
$
```

tr [*options* **]** *chaîne1 chaîne2*

(*Translate*) Copie l'entrée standard sur la sortie standard en effectuant au passage certaines traductions de caractères. Chaque caractère rencontré appartenant à *chaîne1* est remplacé par le caractère de même rang de *chaîne2* (si *chaîne2* est plus courte que *chaîne1* alors elle est rallongée par répétition du dernier caractère). Des options possibles sont

`-d` supprimer tous les caractères appartenant à *chaîne1*

`-s` en sortie, réduire à un seul caractère toute suite de caractères identiques

`-c` transformer tous les caractères qui ne figurent pas dans *chaîne1*.

Dans l'indication des chaînes, la spécification `[`*c1*`-`*c2*`]` signifie les caractères compris entre *c1* et *c2* et `\`*nnn* signifie le caractère de code *nnn* en octal. Exemples : mise en majuscule puis suppression des fins de ligne :

```
tr [a-z] [A-Z] < fic_entree | tr -d '\012' > fic_sortie
```

head [*-nombre* **]** *fichier* **...** *fichier*

Ecrit sur sa sortie standard les *nombre* premières lignes de la concaténation des fichiers indiqués. Par défaut, *nombre* $= 10$.

uniq [*options* **]**

Cette commande détecte dans son entrée standard les *lignes identiques consécutives* et leur applique un traitement défini par *options*. Les options sont :

`-u` copier sur la sortie standard les lignes qui ne font pas partie d'un groupe de plusieurs lignes consécutives identiques

`-d` copier sur la sortie standard une ligne de chaque groupe de plusieurs lignes consécutives identiques

`-c` copier sur la sortie standard chaque ligne, en la faisant précéder du nombre d'occurrences consécutives identiques

Exemple : voir sort ci-dessous.

sort [*options* **]** *fichier* **...** *fichier*

Cette commande trie les lignes des fichiers indiqués, ou l'entrée standard, et écrit le résultat sur la sortie standard. De nombreuses options sont possibles (voir le manuel UNIX) notamment pour spécifier une décomposition en champs des lignes lues. Parmi les principales options :

`-b` ne pas tenir compte des blancs qui sont *devant* les valeurs à comparer

`-d` utiliser l'ordre lexicographique

`-f` ne pas distinguer les lettres majuscules des minuscules

`-n` trier selon la valeur numérique

`-r` trier en ordre décroissant

Exemple : « trouver les trois noms les plus fréquents dans une liste de noms » (Remarquer dans cet exemple une particularité des commandes avec des tubes : la première ligne se terminant par `|`, le Shell comprend que la commande n'est pas terminée et affiche le prompt secondaire `>` pour demander la suite)

```
$ tr [A-Z] [a-z] < fic | sort -b |
> uniq -c | sort -nr | head -3
52 pierre
48 paul
30 jean
$
```

grep [*options* **]** *expression fichier* **...** *fichier*

(*Global regular expression print*) Recherche et affiche toutes les lignes des fichiers d'entrée contenant une chaîne de caractères qui s'unifie avec *expression*, qui est une *expression régulière*, voir 5.2. Parmi les options possibles :

`-c` donner uniquement le *nombre* de lignes qui répondent à la question

`-h` ne pas mettre le nom du fichier devant chaque ligne qui convient

`-i` ne pas distinguer les lettres majuscules des minuscules

`-l` n'afficher que les noms des fichiers contenant au moins une chaîne qui convient

`-n` chaque ligne est précédée de son numéro dans le fichier

diff [*options* **]** *fichier1 fichier2*

(*Differences*) Cette commande recherche les lignes différentes entre les deux fichiers indiqués. Sauf dans quelques cas vicieux, elle trouve les différences minimales (ce qui n'est pas facile à faire). Chaque différence est exprimée de la manière suivante :

repérage de la différence
[< *copie des lignes concernées dans fichier1*]
[> *copie des lignes concernées dans fichier2*]

Le « repérage de la différence » est une expression de l'une des trois formes :

– *n1,n2* c *n3,n4*
(lignes différentes) qui se lit : pour rendre *fichier1* égal à *fichier2* il faudrait que les lignes *n1* à *n2* de *fichier1* soient remplacées par les lignes *n3* à *n4* de *fichier2*

– *n1* a *n3,n4*
(lignes excédentaires dans *fichier2*) qui se lit : pour rendre *fichier1* égal à *fichier2* il faudrait ajouter après la ligne *n1* de *fichier1* les lignes *n3* à *n4* de *fichier2*

– *n1,n2* d *n3*
(lignes excédentaires dans *fichier1*) qui se lit : pour rendre *fichier2* égal à *fichier1* il faudrait ajouter après la ligne *n3* de *fichier2* les lignes *n1* à *n2* de *fichier1*

Des options possibles sont :

-b ignorer les blancs (espaces et tabulations)

-e supprimer la copie des lignes et repérer les différences sous la forme (à peine distincte de la précédente) des commandes qui permettraient à l'éditeur ed de reconstruire *fichier2* en agissant sur *fichier1*.

Exemple. Supposons que `ficV1.c` et `ficV2.c` sont deux versions voisines d'un gros programme source. La commande

```
diff -e ficV1.c ficV2.c > diffV1V2
```

produit un fichier `diffV1V2` qui exprime leurs différences. Pour archiver les deux versions, il suffit de conserver les fichiers `ficV1.c` (gros) et `diffV1V2` (petit). Plus tard, il suffira d'exécuter les commandes ci-dessous pour obtenir à nouveau le fichier `ficV2.c`

```
$ echo "w ficV2.c" >> diffV1V2  # cette commande ed manquait
$ ed ficV1.c < diffV1V2
```

NOTE. Cette propriété est à la base de SCCS (*Source code control system*), l'outil UNIX de gestion des sources, dont le maillon « difficile » n'est autre que la commande `diff`.

3.4 Gestion des processus

ps [*options* **]**

(*Process stats*) Affiche des renseignements sur les processus qui existent dans le système. Une option courante est `-la`, qui produit l'affichage de tous les renseignements (`l`) sur tous les processus (`a`). Exemple d'utilisation : voir page 18 ci-dessous.

kill *-numéro pid*

Envoie le signal correspondant au numéro indiqué au processus ayant le numéro de processus (*pid*) indiqué.

Voici comment les utilisateurs normaux (que nous sommes) utilisent cette commande et la précédente : pour une raison quelconque nous n'arrivons plus à avoir la main au terminal. Les moyens d'interruption ordinaires, `Ctrl-C` (interruption douce) et `Ctrl-\` (interruption forte) se montrent inefficaces. Nous devons alors obtenir le prêt momentané d'un autre terminal sur lequel nous nous connecterons sous notre nom (autrement nous ne pourrions pas agir sur nos processus), par exemple à l'aide de la commande su, afin d'exécuter la commande ps pour obtenir des informations sur les processus (on ne montre que les informations qui nous intéressent ici) :

```
$ ps -gl
UID PID PPID STAT TT COMMAND
...
300 186 1 S tty2 -sh (sh)
300 209 186 R tty2 monprog
...
$
```

le but de tout ceci est d'obtenir le numéro (PID) du processus à tuer. Le numéro d'utilisateur (UID) permet de chercher parmi nos processus. Le processus en question peut être reconnu à travers le nom de la commande (COMMAND) ou bien à travers sa filiation (PPID donne le PID du père). Ici, le processus défectueux a le numéro 209 (commande monprog) ; il est le fils du processus 186 (le *shell* lui-même) qui est un processus primaire (fils du processus 1). On peut le tuer :

```
kill -9 209
```

-9 est le numéro du signal qui tue à coup sûr, il ne peut être ni masqué ni récupéré. Si malgré tout cela ne libère pas le terminal, il ne nous reste plus qu'à tirer sur le *shell* lui-même, en faisant « `kill -9 186` ».

su *username*

(*Super user*) Cette commande permet d'ouvrir une « sur-session » par dessus une session existante. Le temps de quelques manipulations, le propriétaire du terminal est changé. Le mot de passe du nouvel utilisateur sera demandé avant de rendre effectif le changement de propriétaire. On revient à l'état précédent de la même manière qu'on termine une session normale, par une ligne réduite au caractère *Ctrl-D*

commande **&**

Exécution d'une commande en arrière-plan. Le *shell* lance la commande indiquée et, sans en attendre la terminaison, réaffiche immédiatement le *prompt* pour vous permettre d'exécuter d'autres travaux. Bien entendu, la tâche en arrière-plan ne doit pas être interactive, et on a tout intérêt à ce qu'elle soit aussi peu bavarde que possible car ses sorties s'afficheront à l'écran, se mélangeant à celles d'un autre travail en cours. Exemple :

```
$ cc -c monprog.c &
$
```

nice *commande* **[** *argument* **...** *argument* **]**

Exécute la commande indiquée avec la priorité la plus faible

Chapitre 4

Le shell, suite

4.1 Variables

Le système des variables d'environnement permet l'écriture de commandes générales, indépendantes du site, du terminal ou de l'utilisateur qui les emploie, leur concrétisation étant réalisée par les valeurs particulières que ces variables reçoivent pour chaque site et pour chaque utilisateur.

Un nom de variable est fait de lettres et de chiffres et commence par une lettre. Les valeurs des variables sont des chaînes de caractères. On affecte une valeur à une variable par l'expression

variable=*valeur*

(attention : pas de blanc de part et d'autre de =). On fait référence à la valeur d'une variable par l'expression

$*variable*

ou, en cas d'ambiguïté[1]

${*variable*}

Exemple :

```
$ NOM=MoinCarr
$ echo $NOM
MoinCarr
$ echo $NOMVers1
                                la variable NomVers1 n'existe pas
$ echo ${NOM}Vers1
MoinCarrVers1                   la concaténation est correcte
$
```

Variables prédéfinies et fichier profile. Un certain nombre de variables sont prédéfinies par le système, ou bien sont définies par l'utilisateur dans son fichier *profile* (voir ci-dessous) :

`$HOME` le répertoire de travail *initial*, c'est à dire celui qui est défini lors de la procédure de connexion. C'est aussi celui qui est pris lorsqu'on utilise la commande cd sans argument.

`$PATH` la suite de chemins qu'utilise le système pour rechercher les commandes. Ces répertoires sont séparés par « : ». Une définition typique est « `.:./bin:$HOME/bin:/bin:/usr/bin` »

[1] En réalité il y a bien d'autres manières de faire référence à la valeur d'une variable, en donnant des valeurs par défaut, etc... Se référer au manuel UNIX.

`$MAIL`	le nom du fichier qui sert de boîte aux lettres. L'examen de la date de la dernière modification de ce fichier permet à la procédure de connexion d'afficher ou non le message « You have mail »
`$PS1`	le *prompt* primaire. Habituellement $
`$PS2`	le *prompt* secondaire (celui qui est affiché lorsqu'une commande n'est pas terminée sur la première ligne). Habituellement >.
`$TERM`	l'identification du type du terminal utilisé. Il renvoie à la base de données `/etc/termcap` et est utilisé par l'éditeur vi

Le fichier *profile* est un fichier qui doit se nommer « `.profile` » et se trouver dans le répertoire de travail initial (`$HOME`). Il est destiné à personnaliser l'environnement de travail en exécutant un certain nombre de tâches initiales, dont la redéfinition de variables prédéfinies ou la définition de nouvelles variables. Par exemple, voici un fichier *profile* typique

```
date
MAIL=/usr/spool/mail/henri
HOME=/usr/users/henri
PATH=.:./bin:/bin:/usr/bin
TERM=vt100
export MAIL HOME PATH TERM
```

4.2 Enchaînement des commandes

Il existe plusieurs manières d'enchaîner l'exécution des commandes. L'enchaînement le plus simple, inconditionnel, s'obtient en écrivant les commandes sur la même ligne, séparées par le caractère « ; » :

commande1 ; *commande2*

les commandes sont exécutées les unes à la suite des autres, dans l'ordre où elles apparaissent, comme si elles avaient été écrites sur des lignes successives. Exemple :

```
date ; cat /usr/users/infos_du_jour
```

L'*enchaînement conditionnel* utilise le code de retour (*status*) des commandes. Il faut savoir que chaque commande se termine en rendant une valeur numérique, censée représenter le succès ou l'échec de l'exécution de la commande[2]. *La convention est la contraire de celle adoptée en C* : un status nul représente un succès, et la commande est considérée comme « vraie », tandis que toute valeur non nulle représente un échec et la commande sera tenue pour « fausse ». L'enchaînement conditionnel se fait par

commande1 && *commande2*

qui signifie « si l'exécution de *commande1* réussit, alors exécutez *commande2* », et

commande1 || *commande2*

qui signifie « si l'exécution de *commande1* échoue, alors exécutez *commande2* ». On considère souvent qu'un tube

commande1 | *commande2*

[2]Si la commande provient d'un programme C, c'est la valeur s précisée lors de l'appel de la fonction « `exit(s) ;` » ou dans le « `return s ;` » qui correspond à la fonction main.

est aussi une manière d'enchaîner des commandes, mais ce n'est pas tout à fait exact car les deux commandes sont exécutées en parallèle. En tout cas, un tube permet d'utiliser la sortie d'une commande comme *entrée* d'une autre.

Il existe un moyen pour utiliser la sortie d'une commande comme valeur dans n'importe quelle expression, et en particulier comme *argument* d'une autre commande. L'expression

‘*commande*‘

représente le texte écrit par *commande* sur sa sortie standard. Les fins de ligne sont remplacées par des blancs. Exemples :

```
$ echo Bonjour: ‘date‘
$ for f in ‘ls *.c‘ ; do mv $f ${f}_sauve ; done
```

4.3 Scripts

Un *script* est un fichier entièrement fait de commandes, les mêmes que celles qu'on aurait pu composer directement sous le *prompt* du *shell*. On obtient l'exécution de ces commandes en tapant le nom du *script* comme s'il avait été une commande prédéfinie ou un fichier binaire exécutable. Un *script* est un fichier de texte, que vous composez à l'aide d'un éditeur de textes. Mais ce doit être aussi un fichier exécutable ; si vous venez de le créer, avant de l'essayer vous devez changer son mode par la commande « `chmod +x` *fichier* ».

4.3.1 Arguments

Un *script* se comportant comme une commande, il peut posséder ses propres arguments. Dans le *script*, ils peuvent être accédés par les expressions :

`$0` le nom du script

`$1` le premier argument

⋮

`$9` le neuvième argument (et le dernier qui peut être ainsi référencé)

`$*` la liste de tous les arguments

`$#` le *nombre* des arguments.

La commande `shift` décale les arguments d'un cran : `$2` devient `$1`, `$3` devient `$2` etc.

4.3.2 Variables

Un script peut comporter un ensemble de définitions de variables d'environnement. Mais il faut savoir que ces définitions ne seront pas transmises aux scripts appelés, à moins qu'elles ne soient exportées par la commande

`export` *variable ... variable*

4.3.3 Déroutement de l'entrée

Normalement, lorsque le *shell* est en train d'exécuter un script il lit les commandes dans le fichier, mais ces commandes lisent leurs données dans l'entrée standard de la commande qui l'a lancé. Dans un script, l'expression

```
commande << chaîne
```

produit l'exécution de la *commande* indiquée en lui fournissant pour entrées les lignes suivantes du script, jusqu'à la rencontre de la *chaîne* indiquée, qui sert dans ce cas de marque de fin de fichier. Exemple :

```
cat > exemple << _FIN_
Ces trois lignes
deviendront le contenu
du fichier exemple.
_FIN_
```

4.3.4 Structures de contrôle

Plusieurs sortes de structures de contrôle peuvent être utilisées pour sélectionner ou itérer l'exécution des commandes qui composent un script. Nous n'en mentionnerons que trois.

La sélection

```
case chaîne in
  motif1) liste-de-commandes1 ;;
  motif2) liste-de-commandes2 ;;
     ⋮
  motifN) liste-de-commandesN ;;
esac
```

Chaque *motif* peut comporter des caractères génériques. Exemple : on souhaite écrire une commande ajout qui fonctionne de la manière suivante : l'appel

```
ajout fichier1 fichier2
```

concatène *fichier1* à la fin de *fichier2*. L'appel

```
ajout fichier
```

concatène à la fin de *fichier* les lignes tapées à l'entrée standard. Tout autre appel produit l'affichage du message

```
Usage: ajout [ source ] destination
```

Voici le texte de la commande ajout :

```
case $# in
   1) cat >> $1 ;;
   2) cat >> $2 < $1 ;;
   *) echo "Usage: ajout [ source ] destination" ;;
esac
```

Le test dans un script s'obtient par la construction :

```
if condition
then commande1
fi
```

ou

```
if condition
then commande1
else commande2
fi
```

L'expression *condition* est une commande quelconque. La commande *test* permet d'évaluer des expressions logiques. Elle a la syntaxe suivante :

test *expr*

[*expr* **]**

Les blancs avant et après expression sont obligatoires. La commande `test` se termine avec succès si et seulement si l'expression *expr* est vraie. Pour construire cette expression on dispose des options suivantes :

`-d` *nom*	vrai si le répertoire *nom* existe,
`-f` *nom*	vrai si le fichier *nom* existe,
`-s` *nom*	vrai si le fichier *nom* n'est pas vide,
`-r` *nom*	vrai si le fichier *nom* est accessible en lecture,
`-w` *nom*	vrai si le fichier *nom* est accessible en écriture,
`-x` *nom*	vrai si le fichier/répertoire *nom* est accessible en exécution,
ch1 `=` *ch2*	vrai si les chaînes *ch1* et *ch2* sont indentiques,
ch1 `!=` *ch2*	vrai si les chaînes *ch1* et *ch2* sont différentes,
`!` *exp*	*négation* logique d'une expression,
exp1 `-o` *exp2*	*ou* logique entre deux expressions,
exp1 `-a` *exp2*	*et* logique entre deux expressions,

Il est possible de comparer deux nombres avec l'expression *n1 cmp n2* où *cmp* est l'une des options suivantes : `-eq`, `-ne`, `-lt`, `-le`, `-gt` ou `-ge`. Un petit exemple d'utilisation :

```
if [ -f $fichier ]
then
   echo -n "le fichier $fichier existe, "
   if test -x $fichier
   then echo "et il est executable."
   else echo "mais il n'est pas executable."
   fi
else echo "le fichier $fichier n'existe pas."
fi
```

L'itération dans un script s'obtient par la construction :

```
for variable in chaîne ... chaîne
do commandes
done
```

Exemple : La commande *emballer* concatène plusieurs fichiers en un seul, de telle manière que le fichier obtenu est *la commande qui reconstruit les fichiers originaux*. Voici le texte de la commande emballer :

```
for f in $*
do
   echo "cat > $f << 'Fin $f'"
   cat $f
   echo "Fin $f"
   echo "echo fichier $f recree"
done
```

Utilisation :

```
$ emballer machin truc chose > tout
$
```

Nous pouvons examiner le contenu du fichier tout :

```
$ cat tout
cat > machin << 'Fin machin'
ici apparait le contenu du fichier machin
Fin machin
echo fichier machin recree
cat > truc << 'Fin truc'
ici apparait le contenu du fichier truc
Fin truc
echo fichier truc recree
cat > chose << 'Fin chose'
ici apparait le contenu du fichier chose
Fin chose
echo fichier chose recree
$
```

Exécutons le fichier tout :

```
$ chmod +x tout
$ tout
fichier machin recree
fichier truc recree
fichier chose recree
$
```

En examinant notre répertoire de travail nous constaterons que trois fichiers machin, truc et chose ont bien été créés.

Chapitre 5

éditeurs de textes et expressions régulières

5.1 Editeur de textes vi

L'éditeur `vi` est un éditeur pleine page qui utilise au mieux les possibilités des terminaux vidéo (sans toutefois approcher les capacités des interfaces graphiques avec fenêtres, menus déroulants et souris). Il est beaucoup plus facile et agréable à utiliser qu'on ne le dirait au premier abord, et il a le mérite d'être disponible sur presque tous les systèmes UNIX, aussi rustiques ou mal configurés soient-ils.

L'éditeur a besoin de connaître le type du terminal utilisé. Cela se fait au moyen de la variable d'environnement TERM qui renvoie au fichier `/etc/termcap` dans lequel sont définis tous les terminaux connus. Si cette définition pose un problème, le lancement de `vi` produira le message « *using open mode* ».

Déplacement.

→ ou `l`	avancer d'un caractère,
← ou `h`	reculer d'un caractère,
↑ ou `k`	monter à la ligne précédente,
↓ ou `j`	descendre à la ligne suivante,
`:`*nnn*	aller à la ligne *nnn*,
`:$`	aller à la dernière ligne,
Ctrl-F	avancer d'une page,
Ctrl-B	reculer d'une page,

Effacement et « déseffacement ».

`x`	effacer le caractère courant,
`dd`	effacer la ligne courante et la copier dans le tampon
`Y`	copier la ligne courante dans le tampon,
`p`	insérer, à la position courante, le dernier texte copié dans le tampon,
`J`	juxtaposer la ligne courante et la suivante,
`cw`	changer le mot qui débute à la position courante (finir la saisie pas la touche *Escape*).

Insertion. Les commandes suivantes font passer l'éditeur dans un mode « insertion », dans lequel tous les caractères frappés sont insérés tels quels dans le texte. La seule touche sensible est la touche *Escape*, qui fait quitter ce mode.

`i`	insérer avant le caractère courant,
`a`	insérer après le caractère courant,
`A`	insérer à la fin de la ligne courante,
Escape	sortie du mode insertion, retour au mode normal,

Opérations sur les fichiers.

`:r` *fichier*	insérer à la position courante le contenu du fichier indiqué,
`:w` *fichier*	sauver le texte édité dans le fichier indiqué,
`:w`	sauver le texte édité (dans son fichier d'origine),
`:q`	quitter l'éditeur,
`:x`	sauver le texte et quitter l'éditeur,
`:q!`	quitter l'éditeur, même si cela entraîne la perte du texte édité,
Ctrl-L	réafficher l'écran.

Recherche et remplacement.

/*chaîne*/	chercher la chaîne indiquée,
/	rechercher la chaîne précédemment cherchée,
rx	remplacer le caractère courant par le caractère x,
`s...`	substituer une chaîne par une autre (voir la section suivante),
`g...`	opération globale (voir la section suivante),

La plupart de ces commandes peuvent être précédées par un nombre *n* (que l'on tape en aveugle) qui indique que la commande en question doit être effectuée *n* fois.

Toutes les commandes de l'éditeur `ed`, encore plus rustique que `vi`, sont utilisables dans `vi`, il suffit de les faire précéder de « : ». Les commandes `:r`, `:w`, `:q`, etc. en sont des exemples.

5.2 Expressions régulières

Les explications suivantes sont valables pour les éditeurs `ed`, `sed` et `vi`, ainsi que pour la commande `grep`. Dans tous ces programmes, les recherches de chaînes de caractères se font à travers des *expressions régulières*. Une expression régulière est un moyen formel pour spécifier une famille de chaînes de caractères ; utilisée dans une opération de recherche, elle signifie « trouvez la première chaîne du texte qui appartient à la famille que je définis ». Les expressions régulières sont formées avec les éléments suivants :

.	n'importe quel caractère unique.
[$s_1 \ldots s_k$]	ensemble de caractères : **un** caractère parmi ceux spécifiés. Les spécifications s_i sont de la forme c (un caractère) ou c_1-c_2 (un intervalle de caractères).
[^$s_1 \ldots s_k$]	complément : n'importe quel caractère différent de ceux définis par l'ensemble. Le caractère ^ n'a ce rôle que s'il figure en première place, immédiatement après le crochet ouvrant ([).
*expr**	n chaînes de caractères qui appartiennent à la famille définie par l'expression régulière *expr*, avec $n \geq 0$.
expr+	n chaînes de caractères qui appartiennent à la famille définie par *expr*, avec $n > 0$. En clair, au moins une occurrence d'une chaîne.
^*expr*	toute chaîne qui s'unifie avec l'expression régulière *expr* à la condition qu'elle se trouve en début de ligne.
expr$	toute chaîne qui s'unifie avec l'expression régulière *expr* à la condition qu'elle se trouve en fin de ligne. La fin de ligne n'est pas englobée par la spécification.

Nous pouvons compléter maintenant la description de `ed` et `vi`. En `vi`, la commande de substitution est :

`:s/`*expression-régulière*`/`*chaîne*`/`

elle signifie : sur la ligne courante, recherchez la première chaîne qui s'unifie avec l'*expression-régulière* indiquée et substituez-la par la chaîne donnée en deuxième place.

Il est possible de référencer, dans le deuxième champ, des portions de la chaîne reconnue par l'expression régulière du premier champ. Cela se fait avec les symboles « \(» et « \) » :

exp1\(*exp2*\)*exp3*	repère (voir ci-dessous) la chaîne reconnue par l'expression *exp2*.
\n	($n \in \{1, 2, \ldots\}$) apparaissant dans le deuxième champ de la commande de substitution, cette expression designe la $n$ième chaîne repérée dans l'expression régulière du premier champ

Exemple 1. Un fichier contient des dates écrites sous la forme « `date:`m-j-a » (avec $0 \leq a \leq 99$). On souhaite les mettre dans la forme « `date:`j-m-a » (avec $0 \leq a \leq 1999$). On suppose que l'expression « *date :* » est toujours suivie d'une date à transformer. Voici une commande qui fait cela :

```
:s/date:\([0-9]*\)-\([0-9]*\)-\([0-9]*\)/date:\2-\1-19\3/
```

Exemple 2. Il arrive encore aux programmeurs C débutants que nous sommes d'écrire « `if (x = 0)` » à la place de « `if (x == 0)` ». La commande suivante recherche et affiche toutes les lignes de tous nos programmes qui contiennent de telles expressions (dont certaines sont peut-être justes, ici nous ne faisons que les afficher)

```
grep 'if *(.*[^=!<]=[^=].*)' *.c
```

Opérations globales. Il est possible aussi d'effectuer des opérations d'édition sur toutes les lignes d'un fichier ou sur toutes les lignes qui satisfont un certain critère. La forme générale de la commande correspondante est :

n1,*n2*g/*expression*/*commande*

Elle se comprend de la manière suivante : « de la ligne *n1* à la ligne *n2*, sur chaque ligne qui contient une chaîne s'unifiant avec l'*expression* régulière indiquée effectuer la *commande* indiquée ». L'intervalle *n1*,*n2* peut être absent, la recherche porte alors sur tout le fichier. La *commande* indiquée doit être une requête correcte de ed, comme une commande de substitution. Par exemple, la commande

```
g/^chapitre/s/c/C/
```

remplacera (une fois) chapitre par Chapitre dans toutes les lignes qui commencent par ce mot.

Les éditeurs ed et sed. Nous n'avons pas expliqué l'éditeur de textes ed (voir le manuel UNIX). Sa rusticité est telle qu'on ne s'en sert jamais de manière interactive, mais uniquement dans des scripts, pour effectuer des transformations globales de fichiers. Par exemple, on peut imaginer le script suivant, pour remplacer tous les « chapitre » par « Chapitre », les « section » par « Section » et les « alinea » par « Alinea » :

```
ed $1 < FIN         # appel de ed
g/chapitre/s/c/C/
g/section/s/s/S/
g/alinea/s/a/A/
w                   # on sauve le texte modifie
FIN
```

(toutes ces commandes ont été expliquées à l'occasion de `vi`). Cet exemple est correct mais maladroit, car le fichier édité est intégralement parcouru une première fois pour la première commande, puis il est parcouru à nouveau pour la deuxième commande, puis encore une fois pour la troisième. Cette remarque est à l'origine de l'éditeur de textes `sed` : c'est la même chose que `ed`, mais il faut donner en commençant toutes les commandes qu'on envisage d'exécuter (`sed` ne convient pas pour l'édition interactive), ce qui permet d'organiser le travail autrement. La double boucle qui exprime le travail effectué par `ed` dans l'exemple précédent :

```
pour chaque commande C
    pour chaque ligne L du fichier
        effectuer la commande C sur la ligne L
```

est remplacée en `sed` par :

```
pour chaque ligne L du fichier
    pour chaque commande C
        effectuer la commande C sur la ligne L
```

Chapitre 6

Développement des programmes

6.1 Compilation et édition de liens

La principale commande concernée par cette question est la commande cc, qui enchaîne le prétraitement, la compilation et l'édition de liens d'un ou plusieurs fichiers :

cc [*options* **]** *fichier* **...** *fichier*

(*C compiler*) Les *fichiers* indiqués sont soit des fichiers sources, avec des noms terminés par « `.c` », soit des fichiers objets, avec des noms terminés par « `.o` ». Parmi les options peuvent figurer des bibliothèques (*libraries*), éventuellement avec des noms abrégés comme « `-lm` ».

6.2 Compilation

Dans le cas le plus général (des options peuvent altérer ce comportement) la commande cc provoque la compilation de tous les fichiers sources suivies, si ces compilations ont réussi, de l'édition de liens entre les fichiers suivants :

- les fichiers objets produits par la compilation des sources indiqués dans la commande `cc`,
- les fichiers objets indiqués dans la commande `cc`,
- les bibliothèques indiquées (options `-l...`),
- les bibliothèques standard, qu'il n'est a pas obligatoire de préciser.

Exemple : compilation de `pripal.c` et `moicarr.c`, édition de liens des fichiers objets correspondants avec le fichier objet gauss.o, la bibliothèque mathématique et la bibliothèque standard :

```
cc pripal.c moicarr.c gauss.o -lm
```

Parmi les nombreuses options possibles, les plus fréquemment utilisées sont :

`-o` *nom* — donner ce *nom* au fichier final produit qui, en l'absence d'une telle indication, s'appelle « `a.out` ». En présence de l'option `-c`, c'est le fichier objet résultat de la compilation qui est ainsi nommé.

`-c` — supprimer l'édition de liens : ne faire que la compilation des fichiers sources indiqués. Chaque fichier objet produit a le même nom que le source correspondant, avec « `.o` » à la place de « `.c` ». Si un seul source est spécifié, on peut utiliser l'option « `-o` *nom* » pour donner un nom différent au fichier objet produit.

`-D`*nom*`=`*chaîne* — compiler chaque fichier source comme s'il comportait en première ligne la directive « `#define` *nom chaîne* »

`-E` ne pas compiler, se limiter à écrire sur la sortie standard le résultat du prétraitement. Ceci permet de contrôler les *macros*, etc.

`-S` supprimer l'édition de liens et l'assemblage : ne faire que la compilation des sources indiqués, en produisant des objets sous la forme de *sources pour l'assembleur*. Ceci permet de contrôler ou d'améliorer le code produit. Chaque fichier objet produit a le même nom que le source correspondant, avec « `.s` » à la place de « `.c` ».

`-g` ajouter au fichier objet l'information symbolique nécessaire au *debugger*

`-O` optimiser le code produit. Les optimisations faites dépendent du compilateur utilisé. Si on envisage de *debugger* le programme il vaut mieux ne pas utiliser cette option, car elle modifie ou même supprime des instructions et on ne reconnait pas le texte source

Edition de liens. L'éditeur de liens prend un ou plusieurs fichiers objets et une ou plusieurs bibliothèques[1] et produit soit un fichier binaire exécutable, soit un fichier objet destiné à apparaître dans des éditions de liens ultérieures (édition de liens *incrémentale*). Options courantes :

`-l`*nom* abréviation du nom `/lib/lib`*nom*`.a` (*nom* est une chaîne de caractères quelconque). Si cette bibliothèque n'existe pas, alors `/usr/lib/lib`*nom*`.a` est recherchée. Exemples fréquents :

- `-lm` bibliothèque mathématique
- `-ll` bibliothèque `Lex`,
- `-ly` bibliothèque `Yacc`,

`-r` incorporer au fichier produit l'information nécessaire pour qu'il puisse figurer dans une autre édition de liens.

6.3 Maintient de la cohérence

Seuls les programmes correspondant aux petits exercices d'école sont entièrement écrits dans un même fichier. Que ce soit pour des raisons formelles (chaque fichier implante une fonctionnalité clairement définie) ou pour des raisons pratiques (chaque fichier est plus petit, donc plus vite compilé, que le tout) les moyens et gros programmes sont toujours repartis sur plusieurs fichiers sources. Souvent ces sources partagent par la directive « `#include` » plusieurs autres fichiers de déclarations communes. Pendant le processus de développement il est fréquent qu'on modifie un seul fichier, le problème se pose alors de ne recompiler que les fichiers qui doivent l'être, sans faire du travail inutile mais en s'assurant du maintient de la cohérence (i.e. sans oublier des recompilations).

La commande `make` remplit ce rôle. Elle lit un fichier, généralement appelé makefile, où sont décrites les dépendances entre les fichiers qui composent le logiciel et elle génère et exécute les commandes utiles pour remettre à jour un certain fichier cible.

Le fichier `Makefile`. Ce fichier contient une liste de règles de dépendance. Chacune est faite d'un membre gauche, le fichier cible, séparé par : d'un membre droit, la liste des fichiers dont le fichier cible dépend, suivis de la commande qui fabrique le fichier cible. Exemple :

```
interpol : pripal.o moicarr.o gauss.o
  →  cc -o interpol pripal.o moicarr.o gauss.o -lm
```

[1] Les bibliothèques sont des fichiers objets munis d'un index qui permet d'y retrouver et d'en extraire efficacement des éléments particuliers. C'est la commande ar (voir manuel UNIX) qui fabrique une bibliothèque à partir d'un ou plusieurs fichiers objets et d'autres bibliothèques

La ligne sur laquelle est écrite la commande associée doit commencer par un caractère de tabulation (symbolisé ci-dessus par →). Cette règle de dépendance se lit : « `interpol` dépend de `pripal.o`, `moicarr.o` et `gauss.o` et, s'il n'est pas à jour, on doit exécuter la commande `cc -o interpol pripal.o moicarr.o gauss.o -lm` » Un fichier cible est à jour lorsque

- les fichiers dont il dépend sont à jour et
- sa date de dernière modification est supérieure (plus récente) aux dates de dernière modification des fichiers dont il dépend

Exemple. Imaginons un logiciel construit de la manière suivante : le binaire exécutable final doit s'appeler exe. Il résulte de l'édition de liens entre les fichiers objets produits par la compilation de `moduleA.c`, `moduleB.c` et `moduleC.c`. Dans `moduleB.c` et `moduleC.c` figure la directive « `#include "declar.h"` ». Et dans `declar.h` figure la directive « `#include "params.h"` ». Voici un fichier `Makefile` décrivant toutes ces dépendances :

```
exe : moduleA.o moduleB.o moduleC.o
  →  cc -o exe moduleA.o moduleB.o moduleC.o

moduleA.o : moduleA.c
  →  cc -c moduleA.c

moduleB.o : moduleB.c declar.h
  →  cc -c moduleB.c

moduleC.o : moduleC.c declar.h
  →  cc -c moduleC.c

declar.h : params.h
  →  touch declar.h
```

6.3.1 La commande make

make [*options* **]** *cible* **...** *cible*

Lit le fichier makefile (dans le répertoire de travail) et, par l'examen des dates de dernière modification des fichiers concernés, construit et exécute la liste de commandes nécessaires pour mettre à jour les fichiers *cible* indiqués. Si aucune *cible* n'est indiquée, on prend pour cible le membre gauche de la première règle de dépendance. Lors de l'exécution des commandes, la rencontre d'une erreur arrête le travail. Options principale :

`-f` *fichier*	prendre les règles de dépendance dans le *fichier* indiqué à la place de makefile
`-n`	afficher les commandes nécessaires pour mettre à jour le fichier cible mais ne pas les exécuter.

Exemple. Imaginons que, notre programme exe étant à jour, on modifie un détail dans declar.h. Voici comment on refait exe :

```
$ make
cc -c moduleB.c
cc -c moduleC.c
cc -o exe moduleA.c moduleB.c moduleC.c
$
```

Bien entendu, plus le logiciel en cours de développement est important et fragmenté, plus le service rendu par la commande make est important.

Chapitre 7

Références

1. Jean-Paul Armspach, Pierre Colin & Frédérique Ostré-Waerzeggers
   « *UNIX Initiation et utilisation* », InterEditions, 1994

2. Jean-Marie Rifflet
   « *La programmation sous UNIX — 3ème édition* », Ediscience International, 1993

3. Christian Pélissier
   « *Utilisation et administration du système UNIX* », Hermès, 1991

4. Brian Kernighan & Rob Pike
   « *L'environnement de programmation UNIX* », InterEditions, 1986

5. Steve Bourne
   « *Le système UNIX* », InterEditions, 1985

Index