Quelques notions et commandes d'UNIX

Yannick COPIN
y.copin@ipnl.in2p3.fr

v 2.0 – Oct. 2002

Table des matières

1 Introduction

Les systèmes « UNIX »[1] sont caractérisés par leurs aspects :
- *multi-tâche* : plusieurs processus (« programmes ») peuvent être exécutés en même temps ;
- *multi-utilisateur* : plusieurs utilisateurs peuvent avoir accès au système en même temps, indépendamment les uns des autres et avec un partage équitable des ressources (processeur, mémoire, disques durs, etc.) ;
- *orienté réseaux* : le système est adapté à une communication intensive entre ordinateurs.

On n'abordera, dans le cadre de ce document, que les commandes *de base* du système, et leurs options les plus utilisées. Certaines options (p.ex., l'option « `z` » de la commande `tar`) ou même certains programmes (p.ex., `locate`) peuvent ne pas exister sous les systèmes autres que GNU/LINUX, à propos duquel ce document a été élaboré. On ne cherche en aucun cas l'exhaustivité, ni au niveau de la somme des commandes abordées, ni au niveau de chacune de leurs options.

La dernière mise à jour de ce document est disponible à l'adresse suivante : `http://snovae.in2p3.fr/ycopin/info_DEA.html`, et les commentaires – évidemment constructifs ! – sont à adresser à `y.copin@ipnl.in2p3.fr`.

[1] Nom générique regroupant de nombreux systèmes d'exploitation couramment utilisés dans le monde de la recherche, p.ex., GNU/LINUX, SUN/Solaris, IBM/AIX, etc.

1.1 Connexion & déconnexion

Conformément au caractère multi-utilisateur du système UNIX, chaque utilisateur possède un *compte* strictement personnel. Pour accéder au système (*to log in*), il doit donc s'identifier par son « *login* » (nom *unique* sous lequel le système le connaît, p.ex., `ycopin`) et son mot de passe *secret* (un mélange de lettres majuscules/minuscules, de chiffres et de symboles sans signification évidente). Une fois connecté au système, l'utilisateur a accès aux fonctionnalités et fichiers qui lui ont été explicitement autorisés[2].

À la fin de l'utilisation du système, il est nécessaire de se déconnecter (*to log out*) non seulement afin de libérer les ressources, mais également pour des raisons de sécurité – en l'occurrence ne pas permettre l'accès au système à d'autres utilisateurs sous son propre compte. La déconnexion d'un environnement graphique (p.ex., KDE) se fait à l'aide de l'icône approprié, et l'interruption d'une connexion à distance (cf. § 6.1, p.12 pour les commandes permettant les connexions à distance) se fait généralement à l'aide de la commande `exit` ou `logout`.

1.2 Syntaxe des lignes de commande

Les commandes sont des instructions que l'utilisateur fait exécuter au système *via* un interpréteur de commandes (le « *shell* », cf. § 7, p.13 pour un exemple particulier), généralement depuis une fenêtre de type `xterm`, ou éventuellement dans un fichier exécutable regroupant plusieurs commandes appelé « *script* » (cf. § 8, p.18).

La syntaxe standard d'une ligne de commande est la suivante, en prenant pour convention d'écrire entre crochets les parties optionnelles :

```
programme [-options [arguments]] [arguments]
```

`programme` est le nom du programme exécuté par la ligne de commande. Pour l'ensemble des programmes de base du système, il n'est pas nécessaire de spécifier l'adresse du répertoire où les exécutables sont stockés, et le simple nom de la commande suffit (p.ex., `ls`, équivalent à `/bin/ls`). En revanche, les programmes non standards qui ne se trouvent pas dans les chemins décrits par la variable `PATH` (cf. § 7.4.2, p.17) doivent être adressés explicitement pour exécution (p.ex., `~/projet/bin/programme` ou `./programme`).

Les options permettent de modifier, si besoin, le rôle du programme. Ainsi, si `sort` permet de trier les lignes d'un fichier dans l'ordre alphabétique, `sort -r` permet de les trier dans le sens inverse (*Reverse*). Les options courtes sont généralement constituées d'une seule lettre précédée d'un « `-` », et les options dites *longues* d'un `--` (p.ex., `--help` ou `--version`). Plusieurs options courtes peuvent éventuellement être concaténées (p.ex., `ls -al` est équivalent à `ls -a -l`).

Les arguments sont les objets qui seront affectés par le programme (p.ex., `less toto.txt` va afficher le fichier `toto.txt`). Ils peuvent être optionnels s'il existe un argument par défaut.

Attention

Les systèmes UNIX font la différence entre les majuscules et les minuscules : `toto.txt` ≠ `ToTo.TxT`.

1.3 Aide en ligne

La plupart des commandes (p.ex., `grep`) disposent d'une *aide en ligne* décrivant en détail le rôle du programme, l'ensemble des options disponibles, des exemples d'utilisation, etc. Cette aide, qui constitue la documentation de référence des commandes, est accessible par la commande `man` (p.ex., `man grep`), ou de plus en plus par la commande `info` (p.ex., `info grep`).

Conseil

Ne pas hésiter à utiliser l'aide en ligne pour connaître tous les détails d'une commande.

Pour les commandes les plus complexes, il est également intéressant de consulter les manuels et « *tutorials* » du WEB (voir `http://snovae.in2p3.fr/ycopin/info_DEA.html` pour un point de départ).

[2] Il existe un utilisateur particulier, le *super-utilisateur* – ou « *root* » –, pour qui le système est totalement ouvert. Ce compte est réservé à l'administrateur système.

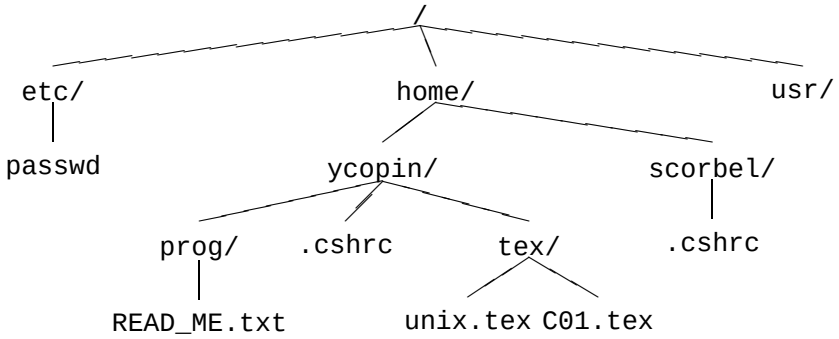

FIG. 1 – Extrait d'une arborescence typique. Les répertoires sont traditionnellement indiqués par un « `/` » final, p.ex., « `etc/` ».

1.4 Chaînes de caractères génériques

L'argument d'une commande est le plus souvent un nom de fichier, de répertoire, etc., c-à-d une chaîne de caractères. Cette chaîne de caractères peut être *explicite* (p.ex., la chaîne de caractères `/etc/passwd` désigne de manière univoque le fichier `/etc/passwd`) ou *générique* (p.ex., la chaîne « `*.c` » désigne l'ensemble des fichiers dont le nom finit par « `.c` », voir ci-dessous).

Certains caractères – appelés *méta-caractères* – ont ainsi une signification particulière dans les commandes et permettent de construire des chaînes de caractères génériques complexes[3]. Concernant les méta-caractères du *shell* (cf. § 7.2.1, p.14 pour plus de détails, et § 4.1, p.7 pour une description des méta-charactères des expressions régulières), les plus courants sont les suivants :

`*` désigne une chaîne de caractères quelconque ;

`?` désigne un caractère quelconque ;

`[...]` désigne un caractère parmi ceux entre crochets, définis par énumération ou par intervalle ;

`[!...]` désigne un caractère *autres* que ceux entre crochets, définis par énumération ou par intervalle.

Par exemple :

`[aeiouy]` désigne une voyelle quelconque,

`[!aeiouy]` désigne un caractère *autre* qu'une voyelle,

`[0-9a-zA-Z]` désigne un caractère alphanumérique quelconque,

`[!A-Z]` désigne tous les caractères *autres* que les majuscules,

> **Exercice 1**
> Que désigne la chaîne générique « `?[aeiuoy]*[!0-9].pas` » ?

2 Système de fichiers

2.1 Généralités

Pour l'utilisateur λ, il existe sous UNIX trois grands types de fichiers : le fichier « normal », le répertoire et le fichier « spécial » (que nous ne considérerons pas ici, p.ex., `/dev/cdrom`). La gestion de ces différents fichiers est assurée par le *système de fichiers*.

Le *fichier normal* contient des données, parfois éditables avec un éditeur de texte, parfois binaires et uniquement lisibles par un programme approprié. Un *répertoire* est une sorte de classeur, dans lequel sont stockés des fichiers normaux afin de mieux organiser l'espace de stockage. Un répertoire peut également contenir d'autres répertoires (sous-répertoires), contenant eux-mêmes des fichiers et/ou des répertoires (sous-sous-répertoires), et ainsi de suite. Ce système hiérarchique prend le nom d'*arborescence* (cf. Fig. 1).

Tous les fichiers sont contenus dans un répertoire père, à l'exception du répertoire « `/` », dit *répertoire racine*, qui constitue le « *père des pères* » des répertoires. À la création d'un répertoire dans l'arborescence, deux sous-répertoires particuliers sont automatiquement créés, le répertoire « `.` », désignant le répertoire lui-même, et le répertoire « `..` », désignant le répertoire père. Cependant, ces répertoires, s'ils

[3] Il faut donc éviter de les utiliser explicitement dans les nom de fichier ou de répertoire.

existent, ne sont pas répertoriés par défaut, de même que tous les fichiers dont le nom commence par « `.` » (fichiers ou répertoires *cachés*).

Pour accéder à un fichier particulier dans l'arborescence, il faut pouvoir l'identifier de façon univoque parmi tous les autres fichiers. Le nom qu'on lui affecte a au maximum 255 caractères, et peut contenir *a priori n'importe quel caractère* (y compris les espaces, mais avec toutefois des réserves quant à l'utilisation des caractères accentués, pas toujours supportés par les systèmes non-francisés). Cependant, pour une utilisation pratique de la ligne de commande, il est préférable que ces noms de fichier ne contiennent pas de caractères spéciaux – p.ex., les méta-caractères du *shell* – pour ne pas poser d'incessants problèmes d'interprétation abusive. En général, on se restreint donc à l'utilisation des majuscules, des minuscules, des chiffres et du symbole « _ ».

Connaissant le nom du fichier (p.ex., le fichier `unix.tex`) et sa position dans l'arborescence, c-à-d l'adresse de son répertoire père (p.ex., le répertoire `/home/ycopin/tex/`), on peut accéder au fichier de deux façon :

le chemin absolu désigne l'adresse du fichier à partir de la *racine*, et *commence donc nécessairement par le caractère* « `/` » ; pour le fichier exemple, on a donc `/home/ycopin/tex/unix.tex`,

le chemin relatif désigne l'adresse du fichier *relativement* au répertoire dans lequel on se trouve au moment de taper la commande (le *répertoire courant*, c-à-d le répertoire « `.` »). Ainsi, si le répertoire courant est le répertoire `/home/ycopin`, le ficher précédent peut être accédé par l'adresse relative `./tex/unix.tex` ou plus généralement, puisque le « `./` » est optionnel, par `tex/unix.tex`[4]. En revanche, si le répertoire courant est le répertoire `prog` de `/home/ycopin`, le même fichier pourra être adressé par l'adresse relative `../tex/unix.tex`. Comme on le voit – et comme son nom le laisse supposer... –, l'adresse relative *dépend* du répertoire courant dans lequel on émet la commande.

Les deux type d'adressage, *absolu* et *relatif*, sont tous les deux parfaitement valides et strictement équivalents : les deux adresses peuvent toujours être utilisées indifféremment dans une commande.

Conclusion

Une adresse absolue commence toujours par « `/` », et une adresse relative par « `./` » (optionnel) ou « `../` ».

Dans l'intégralité de l'arborescence, deux fichiers ne peuvent par définition avoir la même adresse absolue. Ainsi, les fichiers d'un même répertoire ne peuvent évidemment pas avoir le même nom ; en revanche, deux fichiers dans des répertoires différents peuvent être homonymes, puisqu'il n'y a alors pas risque de confusion.

Exercice 2
Savoir repérer l'ensemble des fichiers et répertoires de la Fig. 1 de façons absolue et relative. En particulier, comment repérer explicitement un fichier dans le répertoire courant ?

Les commandes directement liées au système de fichiers sont :

`pwd` affiche le nom absolu du répertoire courant (*Print Working Directory*),

`cd` ***[répertoire]*** change de répertoire (*Change Directory*), de manière relative ou absolue. Sans argument, la commande renvoie au répertoire principal de l'utilisateur (« *home directory* »), tandis que l'argument « `-` » renvoie au répertoire courant précédent, c-à-d le dernier répertoire courant avant le répertoire actuel.

Dans la plupart des *shells*, le caractère « `~` » désigne le répertoire principal de l'utilisateur (*Home directory*), tandis que la chaîne « `~user` » désigne le répertoire principal de l'utilisateur `user`. Ainsi, étant l'utilisateur `ycopin`, la commande `cd ~` est équivalente aux commandes `cd` et `cd ~ycopin`.

Exercice 3
Me trouvant initialement dans mon répertoire principal, où suis-je après la série de commandes suivante : « `cd .. ; cd ; cd - ; cd ~ ; cd .` » ? (les « `;` » ne servent qu'à séparer les commandes, cf. § 7.2.3, p.15)

2.2 Gestion des fichiers

Comme on l'a vu, un répertoire n'est qu'un fichier un peu particulier. La plupart des commandes de gestion des fichiers peuvent donc également s'appliquer aux répertoires, au prix éventuel d'une option supplémentaire. La *récursivité* est la propriété d'étendre la commande à l'ensemble de l'arborescence d'un répertoire (c-à-d à tous ses sous-répertoires, sous-sous-répertoires, etc., et l'ensemble des fichiers de ces répertoires).

[4] Notez dans les deux cas l'absence de « `/` » au début de l'adresse, indiquant qu'il s'agit bien d'adresses *relatives*.

`ls` *[fichier | répertoire]* affiche le contenu d'un ou plusieurs répertoires, ou des renseignements sur un ou plusieurs fichiers (*LiSt*). Sans argument, la commande affiche le contenu du répertoire courant. Si l'argument est un répertoire, elle affiche le contenu de ce répertoire ; si l'argument est un fichier, elle indique si ce fichier existe.

- **`-a`** affiche également les fichiers ou répertoires cachés, c-à-d commençant par « `.` »,
- **`-l`** affiche, en plus du nom, d'autres informations concernant les permissions, le nom du propriétaire et du groupe, la taille, etc. (cf. § 3.2, p.6),
- **`-d`** affiche les répertoires comme des fichiers, plutôt que d'afficher leur contenu,
- **`-R`** affiche le contenu des répertoires de façon récursive,
- **`--color`** indique le type des fichiers par un code couleur.

> **Exercice 4**
> Quel est l'argument par défaut de la commande `ls` ?

`rm` *fichier | répertoire* efface un ou plusieurs fichiers ou répertoires passés en argument (*ReMove*).

> **Attention**
> Il n'y a *aucun moyen* de récupérer un fichier effacé sous UNIX : inutile de faire des yeux de Bambi auprès de l'administrateur système après avoir malencontreusement effacé votre rapport de stage, *il ne peut rien pour vous !* Soyez donc *très attentifs* dès que vous manipulez la commande `rm`, un accident est très vite arrivé... (et n'oubliez pas de faire des sauvegardes de sécurité, cf. § 5.2, p.10)

- **`-i`** demande confirmation avant l'effacement (à imposer de préférence comme défaut, cf. § 7.5.3, p.17),
- **`-f`** *ne* demande *pas* confirmation avant l'effacement (**dangereux**),
- **`-r`** efface un répertoire de façon récursive (**dangereux**).

`cp` *fichier[s] fichier | répertoire* copie un ou plusieurs fichiers, éventuellement en changeant leur nom (*CoPy*), p.ex.,

```
cp ancienne_version.txt nouvelle_version.txt
cp *.c ../archives/
```

- **`-r`** copie un répertoire de façon récursive.
- **`-i`** demande confirmation avant d'écraser un fichier (à imposer de préférence comme défaut, cf. § 7.5.3, p.17),

> **Exercice 5**
> Si le répertoire courant contient les fichiers `fichier.1` et `fichier.2`, que fait la commande « `cp fichier*` » ?

`mv` *fichier[s] fichier | répertoire* déplace un fichier ou un répertoire (*MoVe*), pour changer son nom ou sa position dans l'arborescence, p.ex.,

```
mv mauvais_nom.txt bon_nom.txt
mv *.c sources/
```

- **`-i`** demande confirmation avant d'écraser un fichier (à imposer de préférence comme défaut, cf. § 7.5.3, p.17),

`ln -s` *source [destination]* crée un lien symbolique – c-à-d un raccourci – pour un ou plusieurs fichiers ou répertoires (*LiNk*). Par défaut, la destination est le répertoire courant.

Il existe des commandes spécifiques aux répertoires :

`mkdir` *répertoire* crée un répertoire

- **`-p`** crée les répertoires intermédiaires si nécessaire, p.ex., « `mkdir -p new_dir/new_subdir` ».

`rmdir` *répertoire* efface un ou plusieurs répertoires passés en argument. Ces répertoires doivent avoir été préalablement *vidés* (sinon, utiliser « `rm -r repertoire` », radical mais **dangereux**).

3 Utilisateurs & groupes

3.1 Identité

Dans le cadre d'un système multi-utilisateur, chaque utilisateur est identifié par un nom unique, son « *login* »[5]. De plus, pour permettre le regroupement de plusieurs utilisateurs ayant des besoins communs, il existe des *groupes*, eux aussi identifiés par un nom unique[5]. Ainsi, tous les utilisateurs classiques du système appartiennent au moins au groupe *users*.

`whoami` retourne votre nom d'utilisateur,

`who` retourne le nom des utilisateurs actuellement connectés au système. La commande `who am i` – ou, de façon équivalente, `who mom likes`... – donne de plus amples informations sur votre propre connexion,

`passwd` permet de modifier votre mot de passe (**attention** de na pas oublier le nouveau mot de passe !),

`su` *[nom]* vous connecte temporairement sous un autre nom d'utilisateur. Sans argument, vous connecte en tant que super-utilisateur. Dans les deux cas, vous devez bien évidemment connaître le mot de passe.

3.2 Permissions

Pour respecter l'« intimité » de chacun et éviter les erreurs de manipulation, tous les fichiers et répertoires bénéficient de certaines protections vis-à-vis des différents utilisateurs et groupes. Ainsi, les fichiers critiques du système ne sont accessibles qu'au super-utilisateur, et les fichiers *mails* ne sont lisibles par défaut que par le propriétaire de la boîte aux lettres.

Il existe vis-à-vis de chaque fichier ou répertoire trois types de protection :

le droit en *lecture* permet de lire un fichier ou d'afficher le contenu d'un répertoire,

le droit en *écriture* permet de modifier un fichier ou le contenu d'un répertoire,

le droit en *exécution* permet d'exécuter un fichier s'il s'agit d'un exécutable (programme), ou d'accéder aux fichiers d'un répertoire.

En parallèle, il existe trois types d'utilisateurs :

le propriétaire du fichier (« `u` » pour *User*),

les membres du groupe du propriétaire (« `g` » pour *Group*),

les autres utilisateurs (« `o` » pour *Others*).

et chacune des ces catégories peut accéder à chacune des permissions. L'ensemble des permissions d'un fichier ou d'un répertoire sont affichées dans le premier champ à gauche de la commande `ls -l`, p.ex.,

```
-rw-r--r--   1 ycopin  users        7420 oct 19 22:03 toto.txt

permissions      |     groupe      taille      |        nom
           propriétaire                 date de modification
```

où

`r` désigne une permission en lecture (*Read*),

`w` désigne une permission en écriture (*Write*),

`x` désigne une permission en exécution (*eXecute*),

`-` désigne une absence de permission ;

et où, parmi les 10 champs des permissions,

le 1er champ indique la nature du fichier (« `-` » pour un fichier normal, « `d` » pour répertoire, « `l` » pour lien symbolique, etc.),

les champs 2–4 indiquent les permissions du propriétaire (ici, `rw-` : permission de lecture et d'écriture),

les champs 5–6 indiquent les permissions du groupe (ici, `r--` : permission de lecture),

les champs 7–9 indiquent les permissions des autres utilisateurs (ici, `r--` : permission de lecture).

Les principales commandes concernant les permissions sont :

[5] Plus exactement, chaque utilisateur (resp. groupe) est identifié par un numéro unique, le « *User ID* » (resp. le « *Group ID* »), traduit en *login* (resp. *group*) à l'aide du fichier `/etc/passwd` (resp. `/etc/group`).

chown ***propriétaire fichier*** modifie le propriétaire du fichier ou répertoire (*CHange OWNer*).

-R change le propriétaire d'un répertoire de façon récursive.

chgrp ***groupe fichier*** *idem*, mais pour le groupe (*CHange GRouP*).

chmod change les permissions d'accès d'un fichier ou répertoire. La syntaxe de base est la suivante :

```
chmod [-R] [type_utilisateur][[+|-|=][r|w|x]] fichers
```

où

type_utilisateur est « u », « g », « o », ou « a » pour tous les utilisateurs (propriétaire + groupe + autres).

+|-|= ajoute (« + »), retire (« - ») ou impose (« = ») les permissions considérées,

r|w|x indique le type d'accès que l'on modifie.

Par exemple, la commande « `chmod g+rw fichier` » ajoute les droits de lecture et d'écriture au groupe sur le fichier `fichier`, tandis que « `chmod go-w fichier` » restreint le droit d'écriture au seul propriétaire du fichier.

A priori, vous ne pouvez changer les permissions que des fichiers dont vous êtes propriétaire. Attention donc à la commande `chown`.

> **Exercice 6**
> Que deviennent les permissions du fichier `toto.txt` précédent après la commande « `chmod u+x,g=rw,o-r toto.txt` » ?

4 Exploration

Il existe de nombreuses commandes permettant d'explorer l'arborescence – afin d'y trouver un fichier ou répertoire recherché selon certains critères –, ou le contenu des fichiers. Ces commandes peuvent utiliser les « expressions régulières », que l'on décrit dans un premier temps.

4.1 Expressions régulières

Une « expression régulière » (ou *Regular Expression*, RE) est une chaîne de caractères composée de caractères normaux et de caractères spéciaux, appelés « méta-caractères des expressions régulières ». L'objectif de ces RE est de décrire de façon « standardisée » les chaînes de caractères génériques. Elles sont construites comme des opérations arithmétiques liant entre elles des expressions toujours plus petites.

Malheureusement, les RE ne sont pas complètement standardisées, et différents programmes peuvent utiliser différentes syntaxes. Ainsi, les méta-caractères des RE ne doivent être confondues avec les méta-caractères du *shell*, et doivent au besoin être protégés contre une interprétation prématurée par l'interpréteur de commandes (cf. § 7.2.1, p.14 et l'utilisation des « " »).

Sans entrer dans les nombreux détails et délices du domaine, les principaux méta-caractères des RE sont les suivants :

. désigne un caractère quelconque (sauf le saut à la ligne) ;

***** désigne une répétition quelconque (≥ 0) du caractère ou RE qui précède ;

+ désigne une répétition quelconque (≥ 1) du caractère ou RE qui précède ;

? désigne la présence facultative (0 ou 1 fois) du caractère ou RE qui précède ;

{n} désigne la répétition en nombre *n exactement* du caractère ou RE précédent ;

^ désigne un début de ligne ;

$ désigne une fin de ligne ;

[...] désigne les caractères entre crochets, définis par énumération ou par intervalle,

[^...] désigne les caractères *autres* que ceux entre crochets, définis par énumération ou par intervalle,

| permet de relier deux RE par un « ou » logique,

**** protège le caractère suivant s'il s'agit d'un méta-caractère.

Par exemple :

.* désigne une ligne quelconque, c-à-d n'importe quel caractère – « . » – un certain nombre de fois – « * » – ;

`^$` désigne une ligne vide, c-à-d ne contenant rien entre le début – « `^` » – et la fin – « `$` » – de la ligne ;

`^start` désigne une ligne commençant par la chaîne « `start` » ;

`end$` désigne une ligne se terminant par la chaîne « `end` » ;

`\*.\\` désigne une ligne contenant le caractère « * », suivi d'un caractère quelconque, suivi du caractère « \ » (les méta-caractères « * » et « \ » doivent être protégés par « \ ») ;

`[A-Z][a-z]{9}` désigne une ligne contenant au moins dix lettres successives, la première étant une majuscule, et les neuf autres étant des minuscules ;

> **Exercice 7**
> Comment décrire une ligne commençant par une majuscule ? Ne finissant pas par une minuscule ? Les deux à la fois ?

Pour la construction d'expressions régulières complexes et portables, voir l'utilitaire `txt2regex` (`http://txt2regex.sourceforge.net`).

4.2 Exploration de l'arborescence

`find` recherche des fichiers dans une arborescence selon divers critères, tels que nom de fichier, propriétaire, date de dernière modification, etc. Il s'agit d'une commande très puissante, pouvant s'appuyer sur l'utilisation des expressions régulières (cf. § 4.1, p.7), mais qui peut vite devenir cryptique.

Par exemple :

- La commande `find projet/ -name "*.c" -print` trouve dans le répertoire `projet/` (y compris toute l'arborescence sous-jacente) les fichiers dont les noms (`-name`) finissent par « `.c` » (`"*.c"`), et affiche le nom de ces fichiers (`-print`). Puisque la chaîne générique « `*.c` » doit être transmise telle quelle à la commande `find` sans être interprétée par le *shell*, il est nécessaire de la protéger par des « " ».
- La commande :
  ```
  find . \( -name "*~" -o -name "#*#" \) \
  -follow -mtime +3 -print -exec rm {} \;
  ```
  recherche dans le répertoire courant (« `.` ») les fichiers de nom générique « `*~` » *ou* « `#*#` », y compris ceux accessibles par un lien symbolique (`-follow`), modifiés il y a plus de 3 jours (`-mtime +3`), affiche leur nom (`-print`) et les efface sans confirmation (`-exec rm {} \;`). (Pour une explication des différentes utilisations de « \ », cf. § 7.2.2, p.14).

> **Exercice 8**
> Imaginer des utilisations encore plus compliquées de la commande `find` (exercice facile). Leur trouver une application utile (exercice très difficile).

`locate` ***chaîne_de_caractères*** permet de retrouver l'ensemble des fichiers dont l'adresse absolue contient la chaîne de caractères *explicite* considérée. Cette recherche se fait à l'aide de la liste exhaustive et régulièrement mise à jour de l'ensemble des fichiers du système, et s'avère donc très rapide. Malheureusement, cette commande n'est pas disponible partout.

4.3 Exploration des fichiers

Il existe deux types de fichiers « normaux » : les fichiers *éditables*, conformes à une norme de type ASCII et dont on peut directement visualiser le contenu à l'aide d'un éditeur de texte (p.ex., tous les fichiers sources – C, LaTeX, etc. – directement créés par l'utilisateur), et les fichiers *binaires*, qui ne sont lisibles qu'à l'aide du programme approprié (p.ex., le ficher `.dvi` issu de la compilation d'un fichier source LaTeX, ou encore l'exécutable issu de la compilation d'un fichier source C).

> **Attention**
> La visualisation des fichiers binaires à l'aide des commandes simples (`cat`, `more`, etc.) peut être risquée pour votre terminal. De toute façon, l'utilisateur λ n'a *a priori* pas grand chose à faire avec le contenu des fichiers binaires, qui ont été produit *par* un logiciel *pour* un logiciel.

`file` ***fichier[s]*** détermine le type d'un fichier, et permet en particulier de savoir s'il s'agit d'un fichier éditable. Ainsi, `file unix.*` peut retourner :

```
unix.aux:         LaTeX auxiliary file
unix.dvi:         TeX DVI file (TeX output 1999.10.28:1758)
unix.log:         TeX transcript text
unix.tex:         ISO-8859 text
unix.toc:         LaTeX  table of contents
unix.ps:          PostScript document text conforming at level 2.0
```

Tous les fichiers « `text` » (et certains autres) sont éditables sans risque (mais tous ne sont pas destinés à être effectivement édités, p.ex., les fichiers Postscript).

`cat` ***fichier[s]*** affiche sur la sortie standard – en pratique, « d'un seul coup » – le contenu d'un fichier.

`more` ***fichier[s]*** affiche *page par page* le contenu d'un fichier, et de disposer de quelques outils d'exploration, p.ex., recherche d'une chaîne de caractères. La commande `less`, qui dispose de plus de fonctionnalités et de sécurités, est à utiliser de préférence si disponible.

`q` quitte le programme,

`/string` permet de repérer la chaîne de caractères « `string` » dans le fichier.

`tail` ***fichier[s]*** affiche *la fin* du contenu d'un fichier.

`-n` N affiche les N dernières lignes du fichier (par défaut les 10 dernières lignes),

`+|-num_ligne` affiche le contenu à partir de la ligne `num_ligne` en partant du début (« `+` ») ou de la fin (« `-` ») du fichier,

`-f` relit sans cesse la fin du fichier, dans l'hypothèse où celui-ci est encore en cours d'écriture.

La commande `head`, à la syntaxe très similaire, permet de ne voir que le *début* du contenu d'un fichier.

`diff` ***fichier1 fichier2*** compare deux fichiers et affiche les différences. Telle quelle, cette commande est difficile à décrypter, et il est souvent préférable pour une utilisation pratique d'utiliser une interface, telle que l'outil `compare` d'`emacs`.

`grep` ***chaîne fichier[s]*** recherche une chaîne de caractères (ou une expression régulière, cf. § 4.1, p.7) dans un ou plusieurs fichiers. Cette commande s'avère très utile mais parfois complexe à utiliser.

`-`N affiche non seulement la ligne contenant l'occurence de la chaîne de caractères, mais également les N lignes précédentes et suivantes afin de la replacer dans son contexte,

`-i` ignore les différences majuscules/minuscules dans la chaîne de caractères recherchée,

`-l` ne renvoie que le nom du fichier dans lequel l'occurrence a été trouvée (par défaut, la commande renvoie le nom de fichier, le numéro de ligne et la ligne où l'occurrence a été trouvée),

`-c` retourne, pour chaque fichier, le nombre de lignes où l'occurrence apparaît,

`-v` affiche les lignes qui *ne contiennent pas* la chaîne de caractères spécifiée.

> **Exercice 9**
> Trouver dans le répertoire courant les fichiers C utilisant les variables commençant par « `pixel_` » ou finissant par « `_pixel` », mais *pas* les variables contenant « `_pixel_` ». *Note :* les symboles « `\<` » et « `\>` » correspondent dans une RE respectivement au début et à la fin d'un mot.

4.4 Éditeurs de fichiers

Les deux éditeurs de texte – programmes permettant de lire et de modifier un fichier de type ASCII[6] – les plus répandus du monde UNIX sont `vi` et `emacs` (et leurs dérivés, p.ex., `gvim` et `xemacs`). Il existe cependant un grand nombre d'autres éditeurs, souvent moins puissants toutefois (p.ex., `xedit`, `nedit`, etc.)

4.4.1 `vi`

Il existe deux modes sous `vi`, le mode *Commande*, permettant d'interagir avec le programme, et le mode *Édition*, qui permet l'édition directe du fichier. La touche `Esc` permet de passer en mode *Commande*, et les commandes de base suivantes sont alors disponibles :

`i` passe en mode *Édition*, et ajoute le texte *avant* la position courante,

`a` passe en mode *Édition*, et ajoute le texte *après* la position courante,

[6] À ne pas confondre avec les éditeurs de texte orientés « édition », tels que Word ou FrameMaker.

x supprime le caractère de la position courante,

dd supprime la ligne courante,

:w sauvegarde le fichier,

:wq sauvegarde le fichier et quitte,

:q! quitte sans sauvegarder le fichier.

> **Exercice 10**
> En cas de panique, quel est le plus sûr moyen de sortir de `vi` ?

4.4.2 `emacs`

`emacs` est un autre éditeur de texte, également très puissant. Ne disposant que du mode *Édition*, les commandes sont passées par des combinaisons de touches utilisant « `Ctrl` » et « `Meta` » (généralement la touche « `Alt` »), p.ex., :

Ctrl-k coupe la fin de la ligne à partir de la position courante,

Ctrl-g interrompt la commande en cours,

Ctrl-x + Ctrl-s sauvegarde le fichier,

Ctrl-x + Ctrl-c quitte `emacs`.

Par défaut, `emacs` crée une copie de sauvegarde (*backup*) du fichier édité, en ajoutant à la fin du nom du fichier un « `~` », p.ex., `unix.tex~`.

> **Exercice 11**
> Sous `emacs`, que fait la commande « `Ctrl-e + Ctrl-Espace + M-e + Ctrl-w + Ctrl-v + Ctrl-u + Ctrl-y` » exécutée en milieu de paragraphe ?

5 Gestion des ressources

Puisque plusieurs utilisateurs ont accès au même système, et donc généralement aux mêmes ressources, il est nécessaire de faire attention à ce que ces ressources soient *équitablement* reparties. En ce qui concerne l'espace de stockage (généralement des disques durs), il peut exister un système de « quota », permettant à l'administrateur système de limiter l'espace disponible de chaque utilisateur. Mais de façon générale, il est important que chaque utilisateur fasse lui-même attention aux ressources, en particulier l'espace disque, qu'il consomme, afin de ne pas pénaliser les autres utilisateurs.

5.1 Stockage

du retourne l'espace de stockage utilisé (*Disk Usage*). Sans argument, il s'agit de l'espace utilisé par le répertoire courant et l'ensemble de son arborescence.

- **-k** indique les tailles en kilo-octets,
- **-h** indique les tailles dans les unités les plus adaptées selon les cas,
- **-c** indique la somme des tailles de tous les répertoires et fichiers considérés,
- **-s** n'affiche que la taille total de chacun des arguments.

df retourne l'espace de stockage utilisé et libre pour chacune des partitions du système de fichiers (*Disk Free*). Quand le « *Use%* » d'un disque est de 100%, le disque est plein, et toute tentative de stockage sur ce disque est vouée à l'échec (c'est une cause relativement fréquente de corruption des fichiers).

5.2 Compression & archivage

gzip ***fichier[s]*** permet de compresser un ou plusieurs fichiers. La compression est particulièrement efficace pour les fichiers ASCII, p.ex., pour les fichiers Postscript souvent très volumineux (compression d'un facteur 2 à 3). Le fichier comprimé remplace finalement le fichier d'entrée, et l'extension « `.gz` » est ajoutée au nom du fichier.

gunzip ***fichier[s].gz*** permet de décompresser un fichier « `.gz` » compressé par `gzip`.

tar permet de créer et de manipuler des fichiers d'archive `tar` (historiquement *Tape ARchiver*). Une archive regroupe en un seul fichier (d'extension « `.tar` ») tout ou partie d'une arborescence.
Il existe trois grands modes d'utilisation :

c permet la *création* d'un fichier d'archive,

x permet le *désarchivage* d'un fichier d'archive,

t permet d'afficher le *contenu* d'un fichier d'archive,

et de très nombreuses options :

f permet de passer un nom de fichier comme nom d'archive (par défaut, `tar` lit l'archive sur une bande magnétique, ce qui ne se fait plus guère de nos jours),

v passe le programme en mode « bavard » (*verbose*),

z permet de manipuler un fichier d'archive compressé par `gzip` (extension « `.tar.gz` » ou « `.tgz` »).

Par exemple,

`tar cvzf projet.tgz projet/` va créer (« `c` ») un fichier d'archive (« `f` ») compressé (« `z` ») `projet.tgz` contenant toute l'arborescence à partir du répertoire `projet/`, en affichant tous les fichiers inclus (« `v` »),

`tar xvzf projet.tgz` va extraire (« `x` ») du fichier (« `f` ») compressé (« `z` ») `projet.tgz` l'arborescence précédemment archivé, en indiquant tous les fichiers extraits (« `v` »). En particulier, l'opération créera donc toute l'arborescence à partir du répertoire `projet/` si elle n'existe pas.

> **Attention**
> Si un fichier de l'archive a un homonyme absolu dans l'arborescence, ce dernier sera remplacé (et donc écrasé) lors du désarchivage.

5.3 Processus

Un processus est un programme *en cours d'exécution* (p.ex., le *shell* ou `netscape`), identifié par son « *Process ID* ». Un certain nombre de ces processus sont nécessaires au bon fonctionnement du système, et les autres ont été lancés par l'utilisateur plus ou moins directement. Il y a deux façons de lancer un programme à partir de la ligne de commande :

`commande` lance la commande en *intéractif* (*foreground*), c-à-d que la fenêtre reste mobilisée tant que la commande n'a pas « rendu la main », soit que le programme soit fini, soit que l'utilisateur en ai interrompu l'exécution volontairement. Pour ce faire,

`Ctrl-c` interrompt *définitivement* (et souvent violemment) le processus (**à éviter**),

`Ctrl-s` *suspend* le processus[7], qui peut alors être repris par `Ctrl-q`,

`Ctrl-z` interrompt *temporairement* le processus, et rend la main à l'utilisateur, qui peut alors reprendre le processus en intéractif avec la commande *shell* « `fg` », ou le reprendre en tâche de fond (voir ci-après) avec la commande *shell* « `bg` ».

`commande &` (noter la présence du « `&` ») lance la commande en *tâche de fond* (*background*), c-à-d que l'utilisateur « reprend la main » immédiatement tandis que le processus tourne seul sans interaction directe avec l'utilisateur. La gestion de ces processus se fait alors avec les commandes `ps` et `kill`.

Exercice 12
Pourquoi ne faut-il pas lancer l'éditeur `vi` en tâche de fond ? Comment rattraper un malencontreux « `vi &` » ?

`ps` affiche la liste des processus en cours d'activité (*Process Status*), selon un format plus ou moins riche. Sans option, la commande n'affiche que les processus actifs de l'utilisateur associés au terminal courant, p.ex.,

```
  PID TTY          TIME CMD
26091 pts/3    00:00:00 tcsh
26402 pts/3    00:04:16 emacs
10405 pts/3    00:00:00 ps
```

Le champ `PID` retourne le numéro d'identification du processus considéré, correspondant à la commande `CMD`.

[7] Attention, si vous n'avez pas lancé de commande, « `Ctrl-s` » suspend directement le *shell*, qui doit alors être relancé par « `Ctrl-q` ».

kill *[-signal] pid* envoie un signal d'interruption à un processus dont on connaît le PID. Pour tuer *efficacement* (quoique brutalement...) un processus hors de contrôle, utiliser le signal « 9 ».

> **Attention**
> La tristement célèbre commande « kill -9 » ne doit être utilisée qu'en dernier recours !

top est une version interactive de ps, permettant une gestion en temps réel des processus (touche « k »), un classement des processus selon leur consommation en mémoire (touche « M ») ou en processeur (touche « P »), etc.

at *time* permet de différer l'exécution d'une commande dans le futur, p.ex., pendant la nuit ou le week-end. Ainsi,

```
/home/utilisateur > at midnight
at> gros_programme
at> <EOT>
warning: commands will be executed using /bin/sh
job 2 at 1999-11-02 00:00
```

permet de lancer le programme gros_programme a minuit ce soir-même. Pour terminer l'entrée des commandes, taper Ctrl-d (signal EOT, pour « *End Of Transmission* »). Il existe plusieurs commandes reliées à at, dont :

atq liste les commandes en attente d'exécution (*AT Queue*),

atrm annule une commande en attente d'exécution (*AT ReMove*).

Il existe d'autres commandes permettant de gérer l'exécution des commandes :

batch exécute la commande dès que le système le permet, c-à-d lorsque la charge de travail du processeur est moindre ;

nice abaisse la priorité d'un programme, en laissant ainsi une fraction de son temps aux autres processus en cours ;

crontab permet d'exécuter une ou plusieurs commandes de façon *périodique*.

6 Communication

Le système UNIX est d'emblée orienté réseau, ce qui permet de distribuer les charges et de partager les ressources entre plusieurs machines, séparées de quelques mètres à plusieurs dizaines de milliers de kilomètres. Il dispose donc de nombreuses commandes permettant la communication et le transfert de données entre machines distantes. Nous n'abordons pas ici les commandes permettant de tester la qualité du réseau, qui concernent avant tout l'administrateur système (ping, nslookup, etc.)

6.1 Connexion à distance

hostname -f retourne le nom complet de la machine sur laquelle vous êtes actuellement connecté,

ssh *[utilisateur@]nom_machine [commande]* permet de se connecter sur une machine distante (*Secured SHell*), éventuellement sous un autre *login* que le *login* courant, et éventuellement pour y lancer une commande.

> **Attention**
> La commande sécurisée « ssh » remplace désormais les commandes telnet et rsh, considérées comme vulnérables d'un point de vue sécurité-réseaux.

6.2 Transferts de données

ftp *nom_machine* permet le transfert de fichiers entre machines distantes (*File Transfer Protocol*). Une fois connecté, de manière nominative (c-à-d identifié par son *login* et son mot de passe) ou de manière anonyme (*login* = anonymous, mot de passe générique = son.adresse@email), les commandes suivantes sont disponibles :

bin permet de transférer des fichiers *binaires*.

> **Attention**
> Par défaut, seuls les fichiers ASCII sont transférables (les fichiers binaires sont transférables, mais peuvent être alors corrompus),

get, put ***nom_fichier*** permettent le transfert système distant → système local (`get`) ou système local → système distant (`put`) d'un fichier explicitement nommé (les méta-caractères ne sont pas décodés),

mget, mput ***nom_fichier*** permettent l'utilisation des méta-caractères dans les noms de fichiers,

prompt off permet de transférer les fichiers adressés à l'aide de méta-caractères sans confirmation,

ls, cd, pwd commandent le système *distant*,

!ls, !cd, !pwd commandent le système *local*.

Si disponible, vous pouvez utilisez des outils FTP plus conviviaux, tels que `ncftp`, d'usage beaucoup plus pratique (rappel de commandes, signets, utilisation systématique des méta-caractères, etc.).

sftp ***[utilisateur@]nom_machine*** est la version non seulement sécurisée, mais également améliorée, de `ftp`. Il est donc préférable (et parfois strictement nécessaire) de l'utiliser.

scp ***[[user@]machine :]adresse_fichier [[user@]machine :]adresse_fichier*** copie des fichiers entre machines du réseau (*Secured CoPy*). Il s'agit de la version sécurisée de `rcp`.

Certaines commandes peuvent s'avérer très utiles dans des cas particuliers, p.ex., :

wget permet de rapatrier des pages WEB ou un site entier (option `-r`, à manier avec précaution),

rsync permet de « synchroniser » efficacement deux arborescences distantes, p.ex. entre l'ordinateur au bureau et celui à la maison.

6.3 Impression

L'impression est typiquement une ressource distribuée entre les machines du réseau local : toutes les machines ont un accès équitable à une imprimante unique. L'impression se fait alors par une *demande d'impression* à un processus chargé de gérer en temps réel la connexion avec l'imprimante (le « démon » d'impression `lpd`).

lpr ***nom_fichier*** lance la demande d'impression d'un fichier sur l'imprimante par défaut[8] (*Line PRinter*),

-P*imprimante* définit explicitement l'imprimante sur laquelle envoyer l'impression, p.ex., `lpr -Plaser2 fichier.ps`.

lpq liste les fichiers en attente d'impression sur l'imprimante par défaut, et indique leur numéro d'ordre (*job*)

lprm ***job*** annule l'impression d'un fichier en attente à partir de son numéro d'ordre, si vous en êtes le propriétaire.

Pour une impression plus compact/économe/écologique, il est parfois intéressant d'imprimer non seulement en recto-verso (si l'imprimante le permet), mais également en « deux pages sur une », à l'aide de commandes telles que `psnup`.

7 Un exemple de *shell* : `[t]csh`

Le *shell* est un « interpréteur de commandes », c-à-d un programme qui permet la saisie des commandes à adresser au système, tout en fournissant certaines facilités. Il existe plusieurs *shells*, sensiblement différents par la syntaxe et les fonctionnalités : le *Bourne shell* et le *Bourne again shell* (« `[ba]sh` », *shell* standard du système GNU/LINUX), le *Korn shell*, etc.

On se limitera ici plus spécifiquement au *C shell* (« `csh` ») et au *Thomas C shell* (« `tcsh` »), dont la syntaxe est plus proche du langage C. Cependant, beaucoup des concepts introduits (redirections, variables, etc.) sont génériques à la plupart des *shells*, seuls leurs mise en œuvre pratique peut varier.

[8] L'imprimante par défaut est celle décrite par la variable d'environnement `PRINTER` si elle existe (cf. § 7.4.2, p.17).

7.1 Généralités

Les shells `csh` et `tcsh` sont très similaires et offrent de nombreuses fonctionnalités très utiles :

Édition : `[t]csh` fournit des outils permettant d'éditer la ligne de commande (souvent similaires à ceux de `emacs`), p.ex.,

`Ctrl-a` place le curseur en début de ligne,

`Ctrl-e` place le curseur en fin de ligne,

`Ctrl-k` coupe la ligne à partir du curseur,

`Ctrl-y` colle la (section de) ligne précédemment coupée.

Historique : toutes les lignes de commande sont mémorisées et peuvent être rappelées pour nouvelle exécution directe ou après modification. En particulier, les flèches « ↑ » et « ↓ » permettent de naviguer dans l'historique des commandes (rappel des commandes).

Complétion : `tcsh` fournit en outre des outils permettant de compléter automatiquement le nom des fichiers et des exécutables, à l'aide de la touche « `Tab` » ou de `Ctrl-d`.

Conseil

Puisque vous allez passer un temps certain à taper des commandes, une bonne maîtrise des différents outils de votre *shell* est très utile, et la lecture attentive de son manuel est donc *vivement* recommandée.

7.2 Les symboles du *shell*

Les symboles du *shell* permettent soit de construire des chaînes de caractères génériques (méta-caractères dont la syntaxe est similaires *mais différente* à ceux des expressions régulières), soit de modifier l'interprétation des commandes par le *shell*.

7.2.1 Les méta-caractères du *shell*

Les méta-caractères permettant la construction de chaînes génériques ont déjà été décrits à la § 1.4, p.3. Un rappel des plus utilisés :

`*` désigne une chaîne de caractères quelconque ;

`?` désigne un caractère quelconque ;

`[...]` désigne un caractère parmi ceux entre crochets, définis par énumération (p.ex., `[aeiou]`) ou par intervalle (p.ex., `[a-z]`) ;

`[!...]` désigne un caractère *autres* que ceux entre crochets, définis par énumération ou par intervalle.

7.2.2 Les caractères spéciaux

Outre les méta-caractères, le *shell* dispose d'autres caractères spéciaux, dont :

L'espace est un caractère spécial dans le sens où il est utilisé pour séparer les différentes parties d'une commande. Cela empêche en particulier son utilisation directe dans les noms de fichier.

Le « \ » (*backslash*) a plusieurs significations :

- avant un méta-caractère, il en annule la signification particulière : p.ex., « `\*` » désigne *explicitement* le caractère « * ». En conséquence, il permet de protéger un caractère du *shell* que l'on veut passer explicitement à une commande qui l'interpretera elle-même. En particulier, il permet d'accéder à des fichiers dont le nom contient des caractères spéciaux ou espaces :
  ```
  mv Chapi\ Chapo\ -\ Generique.mp3 Chapi_Chapo_Generique.mp3
  ```
- il permet de scinder une longue ligne de commande :
  ```
  nom_de_programme_tres_long -avec -plein -d_options \
  -et -des arguments bizarres
  ```

> **Exercice 13**
> Décrire les différentes utilisations du « \ » dans la commande suivante :
> ```
> find . \( -name *~ -o -name #*# \) \
> -follow -mtime +3 -print -exec rm {} \;
> ```

Le « $ » permet d'accéder aux variables du *shell* ou aux variables d'environnement (cf. § 7.4, p.16). Il est souvent utilisé avec la commande `echo`, qui permet d'afficher une chaîne de caractères à l'écran, afin de voir le contenu d'une variable, p.ex., « `echo $PRINTER` » permet de déterminer l'imprimante par défaut.

`'`*chaîne*`'` délimite une chaîne de caractères à l'intérieur de laquelle *tous* les caractères spéciaux perdent leur signification, p.ex.,

```
emacs '*** Gagner des $$$$ ***.txt'
```

`"`*chaîne*`"` délimite une chaîne de caractères à l'intérieur de laquelle tous les caractères spéciaux perdent leur signification *à l'exception* des méta-caractères « `'` » et « `$` ».

7.2.3 Interprétation des commandes

Outre l'utilisation de chaînes de caractères génériques, le *shell* permet de composer des commandes complexes à l'aide de caractères spéciaux.

***cmd* &** lance la commande en tâche de fond, et non pas en intéractif (cf. § 5.3, p.11) ;

cmd1* ; *cmd2 sépare deux commandes sur une même ligne (p.ex., « `cd ..; ls` »), les commandes étant alors exécutées l'une après l'autre ;

cmd1* && *cmd2 sépare également deux commandes, mais n'exécute la seconde que si la première a réussi. Ainsi, la commande « `grep URGENT todo.txt && lpr todo.txt` » imprime le fichier `todo.txt` *si et seulement si* il contient la chaîne de caractères « `URGENT` » ;

cmd1* || *cmd2 sépare encore deux commandes, mais n'exécute la seconde que si la première a échoué (p.ex., « `grep URGENT todo.txt || echo "Rien d'urgent!"` ») ;

`{...}` permettent de regrouper un ensemble de commandes et de les exécuter dans le *shell* courant ;

`(...)` permettent de regrouper un ensemble de commandes et de les exécuter dans un *shell* fils.

> **Exercice 14**
> Analyser le résultat des différentes commandes : `echo $HOME`, `echo '$HOME'`, `echo "$HOME"`, `echo "\$HOME"` et `echo \$HOME`, sachant que la variable d'environnement `HOME` correspond à l'adresse du répertoire principal de l'utilisateur.

7.3 Entrées-sorties

Il est possible de rediriger la *sortie standard* (`stdout`) d'une commande, c-à-d le résultat ou les commentaires issus lors de la réalisation de cette commande, ainsi que son *entrée standard* (`stdin`), c-à-d ses arguments. Il est également possible de rediriger la *sortie standard des erreurs* (`stderr`), c-à-d les éventuels messages d'erreurs émis lors de la réalisation de cette commande. Ces redirections permettent de construire des commandes élaborées à partir de commandes simples.

7.3.1 Redirections simples

`` `cmd` `` capture la sortie standard d'une commande pour former un nouvel argument ou une nouvelle commande. Ainsi, la commande « `vi `grep -l main *.c`` » édite tous les fichiers C contenant la chaîne de caractères « `main` » (notez l'option `-l` de `grep`, nécessaire ici pour ne retourner *que* le nom des fichiers contenant la chaîne de caractères recherchée).

`cmd > output.txt` stocke la sortie standard de la commande `cmd` dans le fichier `output.txt`.

> **Attention**
> Si le fichier de sortie existe déjà, il est *par défaut* écrasé. Pour éviter ce comportement, il est possible de définir la variable *shell* `noclobber` (cf. § 7.4.1, p.16).

`cmd >! output.txt` *idem*, mais en écrasant systématiquement le fichier `output.txt`, même si la variable `noclobber` est définie ;

`cmd >> output.txt` stocke la sortie standard de la commande `cmd` *à la fin* du fichier `output.txt` (concaténation) ;

cmd >>! output.txt *idem*, mais en créant toujours le fichier `output.txt` si nécessaire, même si la variable `noclobber` est définie ;

La redirection de la sortie standard permet p.ex., de stocker le résultat d'une longue commande ou d'une commande émettant beaucoup de messages dans un fichier, pour analyse ultérieure.

> **Exercice 15**
> Comment concaténer les fichiers `f.1`, `f.2` et `f.3` dans un unique fichier `f.all` ?

cmd < input.txt prend l'entrée standard de la commande `cmd` dans le fichier `input.txt`, p.ex., `mail ycopin@obs < programme.c` permet d'envoyer le fichier `programme.c` par email à `ycopin@obs` ;

cmd1 | cmd2 utilise la sortie standard de la commande `cmd1` comme entrée standard de la commande `cmd2` (*pipe*, à ne pas confondre avec « `||` »). Ainsi, `ps aux | grep netscape` permet de visualiser l'ensemble des processus `netscape` tournant actuellement.

> **Attention**
> Une redirection en entrée « `<` » est toujours suivie d'un nom de fichier, mais jamais du résultat d'une commande. Pour reprendre le résultat d'une commande comme entrée standard, utiliser le *pipe* « `|` ».

> **Exercice 16**
> Décrire l'action de la commande :
> ```
> find . -not -name ".*" -not -name "*~" -atime +7 \
> -exec file {} \; | grep text | awk -F : '{print $1}'
> ```
> La commande « `awk -F : 'print $1'` » permet de retourner le début d'une chaîne de caractères jusqu'au premier « : ».

7.3.2 Redirections multiples

cmd >& output.txt stocke la sortie standard *et* la sortie standard des erreurs dans le fichier `output.txt`. De même que dans le cas d'une redirection simple, on a également :

cmd >&! output.txt

cmd >>& output.txt

cmd >>&! output.txt

cmd |& cmd2 utilise la sortie standard *et* la sortie standard des erreurs de la commande `cmd1` comme entrée standard de la commande `cmd2` ;

(cmd > log.txt) >& errors.txt stocke la sortie standard de la commande `cmd` dans le fichier `log.txt`, et sa sortie standard des erreurs dans le fichier `errors.txt`.

7.4 Les variables

Il existe deux types de variables : les variables simples, dont certaines, les variable du *shell*, sont utilisées par le *shell* pour modifier sont comportement, et les variables d'environnement, de portée plus large puisqu'elles peuvent éventuellement modifier le comportement des programmes exécutés. Toutes les variables sont accessible à l'aide du caractère « `$` ».

7.4.1 Les variables du *shell*

Elles permettent de contrôler son comportement (cf. le manuel en ligne pour une liste exhaustive), et sont donc souvent très spécifiques au *shell* utilisé. Elles sont initialisées par la commande `set`. Ainsi, le fichier `.cshrc` (cf. § 7.5.1, p.17) peut contenir les lignes suivantes :

```
set ignoreeof # makes it impossible to logout by typing Ctrl-d
set nobeep    # reduces the noise level
set noclobber # prevents output redirection from overwriting existing files
set autolist = ambiguous     # gives list if completion is ambiguous
set prompt = "%B%n:%b%~ %# " # new shell prompt emphasing user
```

7.4.2 Les variables d'environnement

Ce sont des variables de portée plus générales décrivant l'environnement de l'utilisateur (p.ex., le nom de l'imprimante à utiliser par défaut) et utilisées par certains programmes. Elles ne dépendent donc pas du *shell* utilisé. Elles sont définies par la commande `setenv`, et l'ensemble de ces variables peut être visualisé à l'aide de la commande `printenv`.

Ces variables peuvent être très nombreuses ; les plus usitées sont les suivantes :

`PATH` décrit les répertoires dans lequel le *shell* doit chercher les exécutables. Si un programme ne se trouve dans aucun de ces répertoires, il doit être adressé explicitement. Il est également possible d'ajouter des répertoires personnels à la variable `PATH`. Ainsi,

```
setenv PATH  {$PATH}":/home/ycopin/bin"
```

permet d'ajouter le répertoire `/home/ycopin/bin` au `PATH`, et tous les exécutables s'y trouvant peuvent être appelés directement[9].

`DISPLAY` décrit l'écran physique sur lequel l'affichage doit se faire (cf. § 8.2.2, p.19). Les connexions par `ssh` gèrent automatiquement cette variable, et il ne faut alors *surtout pas* la définir explicitement.

`PRINTER` décrit l'imprimante par défaut.

`HOME` est automatiquement affecté de l'adresse du répertoire principal de l'utilisateur.

> **Exercice 17**
> Examiner le résultat de la commande « `printenv` » et essayer d'identifier les différentes variables et leur intérêt.

7.5 Les fichiers de configuration

Il est possible de configurer son *shell* à l'aide de plusieurs fichiers de configuration lus et exécutés au lancement du *shell*. Pour ma part, je préconise l'emploi d'au moins trois fichiers avec `[t]csh` : `.cshrc`, `.login` et `.alias` (les deux premiers sont des classiques, le troisième plus personnel).

7.5.1 Le fichier `.cshrc`

Le fichier `.cshrc` contient des commandes *shell* exécutées par *tous* les *shells* que vous lancez, y compris les *shells* non-intéractifs (p.ex., *shells*-fils). En conséquence, il *ne doit pas* pas produire de message (*output*), mais seulement affecter des variables du *shell* ou d'*environnement* (cf. § 7.4, p.16).

7.5.2 Le fichier `.login`

Le fichier `.login` contient des commandes *shell* exécutées par les *shells* intéractifs que vous lancez, et peut donc produire des messages. Par exemple pour afficher le contenu d'un fichier pense-bête à chaque connexion, ajouter au `.login` les lignes suivantes :

```
if (-e ~/.pense_bete) then   # Test l'existence du fichier
        echo "--- PENSE-BETE ----------------------------"
        cat ~/.pense_bete    # Affichage du contenu du fichier
        echo "-------------------------------------------"
endif
```

7.5.3 Le fichier `.alias`

La commande *shell* `alias` permet de définir de nouvelles commandes ou le comportement par défaut des commandes de base. Ainsi, pour éviter les catastrophes, il est prudent de définir les `alias` suivants :

```
alias rm 'rm -i'
alias cp 'cp -i'
alias mv 'mv -i'
```

de sorte que la destruction des fichiers ne se fasse qu'après confirmation par l'utilisateur.

Pour définir des `alias` plus complexes, la chaîne « `\!*` » permet de reprendre les arguments passés à la ligne de commande, p.ex.,

[9] Pour des raisons de sécurité, il est conseillé de ne pas ajouter le répertoire courant « `.` » dans le `PATH`, tout au moins pas au début de la variable.

```
alias ll 'ls -Flh --color \!* | more'
```

> **Exercice 18**
> Que fait la commande « viz » définie par l'alias suivant :
> `alias viz 'gunzip \!* ; vi 'basename \!* .gz''`
> sachant que la commande `basename` permet de retirer une extension à un nom de fichier ?

Lorsqu'une commande a été « aliasée », il est encore possible d'accéder à la commande d'origine en la précédant du caractère « `bs` » (p.ex., « `\rm` »)

Afin de ne pas avoir à les reféfinir à chaque connexion, l'ensemble des `alias` peut être regroupé dans le fichier `.alias`, qui sera automatiquement lu à l'ouverture de chaque *shell* par l'inclusion des lignes suivantes au `.cshrc` (cf. § 8.1) :

```
if (-e $HOME/.alias) source $HOME/.alias
```

8 Les *scripts*

8.1 Introduction

Les *scripts* sont des fichiers regroupant plusieurs commandes et permettant leur exécution séquentielle. Leur intérêt est, outre l'interêt de stockage et d'exécution automatique, de fournir des éléments de programmation permettant d'automatiser certaines tâches complexes (sauvegardes, édition automatique de fichiers, etc.). Les capacités de *scripting* de `[t]csh` ne sont pas très puissantes en regard de langages plus évolués (p.ex., `tcl`, `perl`, `python`, etc.), mais permettent de résoudre tout de même les problèmes les plus simples.

Il en existe de deux types :

les « faux » *scripts* ne sont en fait que des regroupements dans un même fichier de commandes, p.ex., pour lancer un grand nombre de commandes à la suite les unes des autres sans intervention de l'utilisateur. Les fichiers `.cshrc`, `.login` et `.alias` (cf. § 7.5, p.17) sont des exemples typiques de ce genre de *scripts*. La commande « `source` » permet d'exécuter les lignes de commandes d'un tel fichier, p.ex.,

```
source .alias
source liste_de_jobs_a_lancer_cette_nuit
```

les « vrais » *scripts* sont de véritables programmes indépendants, pouvant utiliser massivement les différentes structures de programmation offertes par le *shell* (`if`, `foreach`, arguments, etc.)[10]. Ces *scripts* doivent être *exécutables* (permission `x`, § 3.2, p.6) et commencent nécessairement par une ligne commençant par « `#!` » indiquant quel *shell* doit être utilisé pour leur exécution, p.ex.,

```
#! /bin/tcsh -f
```

Dans le cadre d'un script (au delà de la 1re ligne), le caractère « # » commence un commentaire qui ne sera pas interprété jusqu'à la fin de la ligne.

8.2 Exemples

8.2.1 Lancement automatique du système X

Les commandes suivantes permettent de lancer automatiquement la session X sous LINUX. Elles pourraient être placées dans le fichier d'initialisation `~/.login`.

```
set wait=3                              # Affectation de la variable
if ('tty' == "/dev/tty1") then          # Test
    echo -n "Starting window manager in $wait seconds (Ctrl-c to abort):  "
    @ i = $wait
    while ($i > 0)                      # Compte à rebours
        echo -n "\b$i"; sleep 1         # affichage et pause d'une seconde
        @ i--                           # decrémentation de la variable i
    end
    echo "\bGo!"
    startx                              # Lancement de la commande
endif
unset wait                              # Libération de la variable
```

[10] Il est à noter que ces structures sont également accessibles hors-*script*, p.ex., le très utile `foreach`.

8.2.2 Affectation automatique du DISPLAY

Les commandes suivantes permettent de définir automatiquement la variable d'environnement `DISPLAY` ; elles pourraient être placées dans le fichier d'initialisation `~/.login`.

Attention
Cet exemple n'a qu'un caractère pédagogique, puisque les connexions `ssh` gèrent *automatiquement* la variable d'environnement `DISPLAY`.

```
set whoami = ( ‘who am i‘ )
if ( $#whoami == 6 ) then
    set rhost = ‘expr $whoami[6] : ’(\([^)]*\)’‘
    if ( "${rhost}" !~ [1-9]*.[1-9]*.[1-9]*.[1-9]* ) \
        set rhost = "${rhost:r}"
    if ( $?DISPLAY == 0 ) setenv DISPLAY ${rhost}:0
endif
```

8.2.3 Autres exemples

killall

```
#! /bin/tcsh -f
# killall command

set line = (‘ps -ax | grep "$1" | grep -v grep‘)
if ( $#line > 1 ) then
    set pid = ‘expr $line[1]‘
    echo $1 \(pid $pid\) will be killed...
    kill -9 $pid
else
    echo No such process to be killed...
endif
```

mmv

```
#!/bin/tcsh -f
# mmv old_name new_name

if ( $#argv <= 1 ) then
   echo ’you must specify 2 files’
   exit
endif

foreach  name ( $argv[1].* )
   echo $name
   mv -i $name $argv[2].$name:e
end
```

Exercice 19
Quel est le but de ces *scripts* ?

Exercice 20
Écrire un script *shell* permettant d'archiver l'ensemble des fichiers ASCII du répertoire courant et de façon récursive.

Index des symboles & commandes

Solutions aux exercices

1 « `?[aeiuoy]*[!0-9].pas` » désigne l'ensemble des fichiers dont le nom de base (hors extension) ne se termine pas par un chiffre, dont la deuxième lettre est une voyelle, et dont l'extension est « `.pas` ».

2 Le fichier « `READ_ME.txt` » du répertoire courant (« `.` ») peut être repéré explicitement par l'adresse relative « `./READ_ME.txt` ». Ce type d'adressage est utile pour exécuter un programme local quand le répertoire courant n'est pas dans le `PATH` (cf. § 7.4.2, p.17).

3 À nouveau dans le répertoire principal.

4 Puisque la commande « `ls` » retourne le contenu du répertoire courant, l'argument par défaut de `ls` est « `.` ».

5 *A priori* une grosse bêtise, en écrasant le `fichier.2` avec le fichier `fichier.1`, puisque la commande est interprétée par le *shell* comme « `cp fichier.1 fichier.2` ». En revanche, la commande « `cp fichier.* archives/` » est tout à fait sensée, et copie les deux fichiers dans le sous-répertoire `archives/`.

6 Les permissions passent de « `rw-r--r--` » à « `rwxrw----` ».

7 « `^[A-Z]` », « `[^a-z]$` » et « `^[A-Z]*[^a-z]$` »

9 `grep -l "\<pixel_.*\|.*_pixel\>" *.c *.h`

10 Presser calmement la touche « `Esc` », puis taper sereinement la séquence « `:q!` ». Respirez.

11 Rien de bien intéressant...

12 Puisque `vi` est un éditeur de texte, il demande une forte intéraction avec l'utilisateur, et ne doit donc être lancé qu'en mode intéractif. La commande « `vi &` » – qui lance `vi` en tâche de fond – va donc automatiquement être suspendue, et pourra être reprise en mode intéractif à l'aide de la commande *shell* « `fg` ».

15 À l'aide de la commande :

```
cat f.1 f.2 f.3 >! f.all
```

ou encore

```
cat f.[123] >! f.all
```

Noter que, dans tous les cas, le fichier `f.all` est écrasé (présence du « `!` »).

16 Cette commande permet de trouver les fichiers éditables du répertoire courant (hormis les fichiers cachés et les sauvegardes `emacs`) accédés il y a moins d'une semaine. En effet, `find` recherche dans le répertoire courant (« `.` ») les fichiers dont le nom ne commence pas par « `.` » (fichiers cachés) ni ne finit par « `~` » (fichiers de sauvegarde de `emacs`), accédés il y a plus de 7 jours (« `-atime +7` »), et examine ces fichiers à l'aide de la commande « `file` ». Du résultat de la commande précédente, la commande `grep` ne conserve que les lignes contenant la chaîne de caractères « `text` », correspondant aux fichiers éditables. La commande « `awk` » permet d'extraire le nom de fichier recherché du résultat de la commande `file`, à savoir une ligne du type :

```
nom_de_fichier: description du fichier
```

18 Elle permet de `gunzip`er et d'éditer sous `vi` les fichiers initialement compressés.

19 Le premier permet de `kill`er un processus non pas à partir de son `PID`, mais à partir du nom du programme. Le second permet de déplacer plusieurs fichiers dont seule l'extension change simultanément. Ce sont des versions « maison » des commande `killall` et `mmv`.