Tables des matières

Tables des matières

1 Cours Unix utilisateur avancé

Cours IDRIS :
Unix_u

Version 2.2.6 , Version PostScript, Version PDF, Version HTML (http://www.idris.fr/data/cours/unix/user/choix_doc.html) .

Jean−Philippe Proux

2 Introduction

Ce cours a pour objectif de faire découvrir le système d'exploitation unix et de former le lecteur à sa pratique. Ce cours permettra ainsi à chacun d'être plus efficace dans son travail sous Unix en utilisant les commandes appropriées. A la fin du cours le lecteur pourra être considéré comme un utilisateur averti.

2.1 Pourquoi se mettre à Unix ?

Quelques bonnes raisons pour se mettre à Unix

1. En dehors des systèmes Windows je suis perdu ! Toutes les machines, du super−calculateur au PC en passant par les stations de travail, sont sous un système d'exploitation de la famille Unix. Il est donc indispensable, si je veux utiliser convenablement l'environnement des machines mises à ma disposition, d'en connaître un minimum !
2. Dans mon entreprise, mon laboratoire, mon université, mon école, les stations de travail ou de calcul (HP, Sun, IBM, Compaq, SGI, etc.) sont sous Unix et le nombre de PC sous Linux est croissant. En sachant utiliser Unix, je peux travailler de manière identique et efficace sur toutes les plates−formes non Windows.
3. Certaines utilisations nécessitent des contraintes de production fortes telles que :
    - ♦ la disponibilité (pas de reboot, pas d'arrêt),
    - ♦ la performance en charge (nombre d'utilisateurs, de processus),
    - ♦ la pérennité (car Unix est basé sur des standards),
    - ♦ et la stabilité (pas ou peu de bug système).

Une des principales difficultés d'Unix c'est son côté ligne de commande un peu démodé demandant un minimum d'investissement avant de pouvoir faire la moindre tâche. Ce type d'interface reste pourtant inégalé depuis 30 ans ! Il existe depuis plus de dix ans des interfaces graphiques comparables au système Windows, maintenant les environnements graphiques sous Linux n'ont rien à envier aux systèmes de Microsoft.

Sur un système d'exploitation, on ignore en général souvent ce qui se passe "derrière" chacune des actions effectuées. Si ça marche, tout va bien. Pourquoi chercher plus ? Cette ignorance peut être sans conséquence sur le travail quotidien jusqu'au jour où l'ingénieur système change une brique du système (via un service pack ou une mise à jour), modifie un logiciel, ajoute une fonctionnalité ; jusqu'au jour où un disque dur de votre station "crash" et que vous devez en quelques heures changer d'ordinateur et/ou de compte et surtout continuer à travailler.

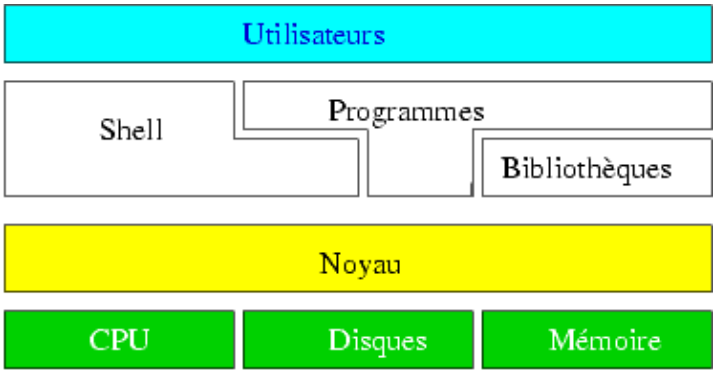
Autre exemple, vous avez l'habitude de rajouter/supprimer quelque chose à la main dans un ou deux fichiers chaque jour, aucun problème ! Comment faire si, pour une raison ou une autre, vous avez 500 fichiers à traiter et que la situation soit urgente (fin de thèse, papier/rapport à renvoyer rapidement etc.) ?

Unix est un des rares systèmes permettant de résoudre l'ensemble des problèmes cités plus haut. Vous avez la possibilité d'installer, de tester, d'utiliser sur de multiples plateformes un système pérenne, ouvert et sans réelle limitation.

2.2 Description et caractéristiques d'unix

- Système ouvert (pas de code propriétaire ; seules certaines implémentations sont propriétaires).
- Multitâches (plusieurs programmes peuvent s'exécuter en même temps, sans blocage).

- Mémoire protégée (pas d'interaction entre les programmes) et virtuelle (le système peut utiliser plus de mémoire que la mémoire physique disponible).
- Multi−utilisateurs (plusieurs utilisateurs travaillent sur la même machine en même temps), gestion des droits.
- Interactif et *batch*.
- Interface graphique X et shell (interpréteur de commande).
- Plusieurs centaines d'outils (manipulation de texte, développement de logiciels, communication etc.).



Sur un système unix, on trouve deux types de personnes, celles qui vont utiliser le système et celles qui vont l'administrer (root). Les premières ont le droit d'exécuter certaines commandes propres à leur environnement et leur travail, quelques commandes liées au système leur sont interdites. Seuls les administrateurs peuvent installer et configurer. Ils sont chargés de la bonne marche de la machine.

3 La connexion en local ou à distance

Dans tous les cas il vous faudra obligatoirement un login/password certifiant que vous avez les droits nécessaires pour vous connecter sur la machine.

En fait vous avez toujours plus.
Pour chaque compte vous avez :

- un login (uid),
- un mot de passe associé,
- un groupe (gid),
- un Home (répertoire de travail),
- un langage de commandes (shell).

Dès lors que vous êtes en local, vous pouvez saisir votre login puis votre password sur l'écran (attention unix fait la différence entre majuscule et minuscule). Deux cas peuvent arriver, soit un écran noir avec un simple prompt ">", soit un environnement graphique avec une gestion de la souris et des fenêtres suffisamment explicites pour démarrer.

Pour la connexion à distance, vous avez bien sûr besoin d'être déjà sur un ordinateur unix/windows, un terminal X ou un simple minitel et d'établir un lien vers la machine unix cible. Chaque type de connexion dépend de la plateforme d'origine. Prenons deux exemples :

- soit vous souhaitez juste avoir une fenêtre texte, pas trop de problème : il suffit d'utiliser la commande telnet (disponible sous les deux environnements, windows ou unix) qui vous invitera à rentrer votre login/mot de passe,
- soit vous voulez avoir du multi fenêtrage : en Unix pas de soucis un telnet/rlogin avec un export de l'affichage suffit. Si vous êtes sous windows, c'est plus dur puisqu'il vous faudra acheter un émulateur X (idem sous Mac). C'est la même chose si vous voulez avoir un environnement windows sous unix !

Remarque : pour sortir d'une session, il est impératif d'utiliser les procédures de déconnexion, en effet unix (comme d'autres systèmes) a besoin de sauvegarder certaines données (flush des buffers via la fermeture des fichiers), démontage des disques pour forcer une mise à jour des fichiers systèmes, etc. Aussi si vous faites un "power off" d'une machine unix, vous risquez d'endommager les fichiers sur les disques. Vous devez faire un `exit` pour revenir à la fenêtre primaire, puis éventuellement un `halt` pour arrêter le système (si vous êtes autorisé à le faire).

4 Le démarrage, un mystère ?

Donc, vous voilà connecté à une machine Unix sous X. Que s'est−il passé au démarrage de votre session ?

1. Sachez d'abord que vous êtes sous votre HOME. C'est un espace disque qui vous appartient, à vous et à vous seul (voir "droits" plus loin). Normalement vous pouvez écrire et lire tous les fichiers qui s'y trouvent. Vous êtes responsable de tout ce qui s'y passe (piratage, saturation des espaces disques, etc...). En général, cet espace est accessible par le chemin suivant */home/groupe/login1* (dans le cas où de multiples groupes existent sur la machine et que login1 soit votre login). Le caractère ~ remplace avantageusement cette chaîne. Il permet également de référencer un HOME quelconque *~login2* par exemple et donc d'éviter de taper le chemin complet (*/home/groupe/login2*).

   Remarque : si vous êtes *login1*, ~ et *~login1* sont identiques.

   Si vous êtes perdus, pour connaître le répertoire courant, utilisez la commande *pwd*, et si vous voulez savoir qui vous êtes, utilisez la commande *id*.

2. Le choix d'un shell peut conditionner une grande partie de votre travail. Le shell est simplement un pseudo langage qui interprète vos commandes. Il va également déterminer les fichiers responsables de la mise en place de votre environnement. Il se nomme ainsi (coquille) car il enveloppe le noyau Unix, toutes les commandes sont passées au noyau à travers votre shell).

   Pour le choix d'un shell, deux grandes familles s'affrontent : les shell−iens et les cshell−iens : la guerre dure depuis longtemps mais est en passe d'être gagnée par les premiers ! Comme vous êtes débutant, je vous conseille de vous rallier au futur vainqueur, c'est−à−dire les shell de la famille du sh (*sh*, *ksh*, *bash*, *zsh*), l'autre famille csh étant uniquement représentée par le *tcsh* et le *csh* lui−même.

   Sur votre station ou votre PC sous Linux vous avez certainement le choix entre le *bash*, le *ksh*, le *tcsh* et le *zsh*. Je conseille donc naturellement le shell *ksh* qui possède le plus grand nombre de fonctionnalités communes. Le *ksh* étant compatible avec le *zsh*, nous allons donc maintenant baser la suite du cours entièrement sur le *ksh* (les différences entre ces shells sont extrêmement minimes).

   Les conséquences d'un tel choix ne sont pas anodines. En effet, votre séquence de "boot" (connexion) est contrôlée par 1 ou 2 fichiers de démarrage. Un premier fichier est exécuté (avant que vous n'ayez la main) c'est le *.profile*. Ce fichier se trouve dans votre HOME et est précédé par un point (lui permettant de rester "caché" si l'on utilise un simple *ls*). Ce fichier sert essentiellement à positionner, par défaut, toutes les variables d'environnement utiles au bon fonctionnement de votre session de travail comme le positionnement des chemins (path) par défaut. Il sert également à définir (via la variable *ENV* généralement valorisée par *ENV=.kshrc*) le nom du fichier qui sera exécuté juste après le *.profile* et qui sera lancé à chaque nouvelle fenêtre, sous−shell ou script (uniquement si cette variable a été exportée voir la suite du cours).

   Le fichier *.kshrc* permet de rajouter des fonctionnalités telles que la définition d'alias (que l'on utilisera plus loin) ou de fonctions.

   A noter que, quel que soit le shell utilisé, tous les environnements sont initialisés par le fichier */etc/profile* modifiable uniquement par l'administrateur.

Le nom du *.profile* et du *.kshrc* peuvent changer suivant les shells et leurs implémentations. Pour le *bash* sur linux les fichiers se nomment *.bash_profile* et *.bashrc*. Dans tous les cas voir le man de votre shell !

5 Où trouver l'information ?

Trouver l'information n'est pas un problème, le plus dur est de se poser la ou les bonnes questions. Deux commandes existent en local *man commande_inconnue* et *news*.

- La première, *man*, permet de tout connaître sur une commande ou un produit sous Unix (comme sa syntaxe ou ses options). En utilisant l'option *−k* vous pouvez chercher un mot clé particulier plutôt qu'une commande. Si la commande n'a pas de man, essayez les options *−?* ou *−h* à l'appel de votre commande.

  Comment lire les syntaxes ?

  - Les [ ] indiquent un argument facultatif. Par exemple un login ou une machine par défaut.
  - Le a | b indique un choix entre a et b .

  ---

  cmd [[login@]machine] [−a] fic[.ps|.gif]

  Cette syntaxe signifie que *cmd* peut être appelée de trois fois six manières différentes.

  - *cmd fic*
  - *cmd −a fic*
  - *cmd machine fic*
  - *cmd machine −a fic*
  - *cmd login@machine fic*
  - *cmd login@machine −a fic*

  En général une commande unix admet une syntaxe composée de :

  - *−option* : option booléenne,
    ls −l
  - *−option[=]valeur* : option avec valeur,
    split −b 10k
  - *fichier* : chemin d'accès à un fichier,
    ls fic
  - ou les deux ou trois à la fois !
    ls −l fic

- La deuxième, *news*, est très utilisée comme canal d'information des nouveautés vers les utilisateurs. Pour l'utiliser, vous avez juste à taper : *news* (si rien ne se passe, vous avez probablement déjà tout lu !)

Évitez de faire "autrement" en cas de difficulté. Ne pas comprendre quelque chose n'est pas grave, ne pas chercher à comprendre l'est plus. Bien sûr disposer de 10 minutes à chaque difficulté n'est pas simple. Un seul grain de sable peut gripper toute la mécanique des systèmes d'exploitation et avoir des conséquences fâcheuses sur vos fichiers ou programmes et donc sur votre travail.

Dans la suite du cours, je ne parlerai que des commandes indispensables accompagnées des options les plus utiles. Si vous souhaitez la liste exhaustive des options de chaque commande, n'hésitez pas à consulter le man

après avoir listé */usr/bin* et */bin*.

6 Tout faire avec et grâce aux fichiers

Un fichier peut avoir 4 statuts, il peut représenter tout d'abord des données ou un programme, il peut aussi représenter un répertoire (d) ! (c'est un fichier rempli d'index), il peut être également un lien symbolique (l), c'est à dire pointer sur un autre fichier (même plus haut dans l'arborescence, cycle possible), et enfin posséder un statut particulier (device) lorsqu'il permet d'accéder à un périphérique (disque, carte son etc.). Pour simplifier, nous nous intéresserons juste aux deux premiers dans ce cours, de plus nous appellerons répertoire le fichier représentant un répertoire !

Remarque sur le nom des fichiers,

- un fichier qui commence par un / est dit absolu, il est nommé en partant de la racine et en descendant dans les répertoires suivants.
- un fichier qui ne commence pas par un / est dit relatif, il est recherché à partir du répertoire courant.
- il n'y a pas de limitation sur le nom des fichiers (à part / bien sûr).

Tous les fichiers, quels que soient leurs types, sont sur des systèmes de fichiers (*file system*). Attention, chaque *operating system* a son propre *file system* et donc son propre formatage (surtout les disquettes !)

- Dos : FAT16,
- Windows9X : FAT 16, FAT 32,
- NT : NTFS,
- Mac : HFS,
- Linux : ext2, ext3 (et tous les autres : journalisé, parallèle, raid etc.).

6.1 ls

Sous Unix, la commande la plus utilisée est, sans aucun doute, *ls*. Elle liste les fichiers et répertoires de votre répertoire courant. Très souvent les fichiers cherchés sont les fichiers sur lesquels on travaille actuellement ou ceux récemment accédés. Nous n'avons donc aucun intérêt à les avoir par ordre alphabétique (par défaut), mieux vaut qu'ils soient classés directement par ordre chronologique croissant. L'ordre croissant est important, il permettra d'avoir les plus récents en dernier. En effet si vous avez plusieurs dizaines de fichiers, le résultat de la commande ne tiendra pas sur une seule page écran et vous serez obligé de jouer avec l'ascenseur de votre fenêtre pour voir ce qu'il y avait au début et cela, à chaque commande *ls* passée ! De plus, il est souvent indispensable de connaître les droits associés aux fichiers, ne serait−ce que pour savoir si c'est un fichier ou un répertoire. Aussi pour faire tout ça d'un coup, une seule commande *ls −lrt*.

Si vous trouvez que c'est trop long, le Korn shell offre la possibilité de créer des alias. Voilà un alias de la commande *ls −lrt* qui peut ainsi être appelée par *ll* en procédant ainsi :
alias ll='ls −lrt'
Cette commande peut être ajoutée dans votre fichier *.profile*.

Pour connaître la liste des alias faites simplement *alias* .

```
Ma_machine>ll
-rw-r--r--   1 jpp user  1372160 dec 14 14:05 htmldoc-1.6-source.tar
drwxr-xr-x   3 jpp user     1024 dec 14 14:19 htmldoc-1.6
drwxr-xr-x   6 jpp user     1024 dec 22 13:26 lirc
drwx------   5 jpp user     1024 dec 28 11:37 Desktop
```

```
drwxr-xr-x   5 jpp user     1024 dec 28 17:06 john-1.6
drwxr-xr-x   2 jpp user     1024 jan 11 16:17 bin
-rw-r--r--   1 jpp user    25861 jan 12 13:04 ref.html
-rw-r--r--   1 jpp user    70144 jan 14 12:22 poster.tar
drwxr-xr-x   2 jpp user     1024 jan 14 12:46 poster
-rw-r--r--   1 jpp user   575186 jan 15 09:46 vitrail12.tiff
-rw-r--r--   1 jpp user   815042 fev  4 17:16 pcmcia-cs-3.0.8.tar.gz
-rw-r--r--   1 jpp user      620 fev  5 13:52 Xrootenv.0
-rw-r--r--   1 jpp user   114525 fev  9 09:23 lirc-0.5.4pre8.tar.gz
-rw-r--r--   1 jpp user   624640 fev 12 10:57 nets-2.0.tar
drwxr-xr-x   3 jpp user     1024 fev 12 10:59 usr
-rw-r--r--   1 jpp user 13953173 fev 16 08:28 imp3.ps
-rw-r--r--   1 jpp user     6774 fev 16 16:25 sondage.html
Ma_machine>
```

Si vous essayez de passer des options/variables à un alias, ce n'est pas possible, vous devez alors utiliser une fonction.

6.2 cp

La seconde commande plus utilisée est celle qui permet de copier un fichier vers un autre fichier ou vers un répertoire (le fichier originel restant en place). Si l'option *-R* est précisée, il est possible de recopier un répertoire (et ses sous répertoires) dans un autre répertoire.

```
Ma_machine>cp fic1 fic2
Ma_machine>cp fic1 rep
Ma_machine>cp -R rep1 rep2
```

6.3 mv

`mv` permet de changer le nom d'un fichier ou de le déplacer vers un autre répertoire.

```
Ma_machine>mv fic1 fic2  #(fic1 n'existe plus)
Ma_machine>mv fic1 rep  #(fic1 existe, mais sous rep)
```

6.4 rm

`rm` permet de supprimer un fichier, ou un répertoire si l'option *-r* est précisée. Pour éviter les confirmations multiples l'option *-f* sera très utile.

```
Ma_machine>rm -f fic*
Ma_machine>rm -r rep
```

Attention *rm*, supprime réellement le fichier, il n'y a pas de restauration possible après un *rm*.
Attention également au *rm toto** mal écrit, un doigt qui fourche, qui glisse pour faire un *rm toto **.

6.5 cd

L'autre commande allant souvent avec *ls* est *cd*. Elle permet de changer de répertoire.

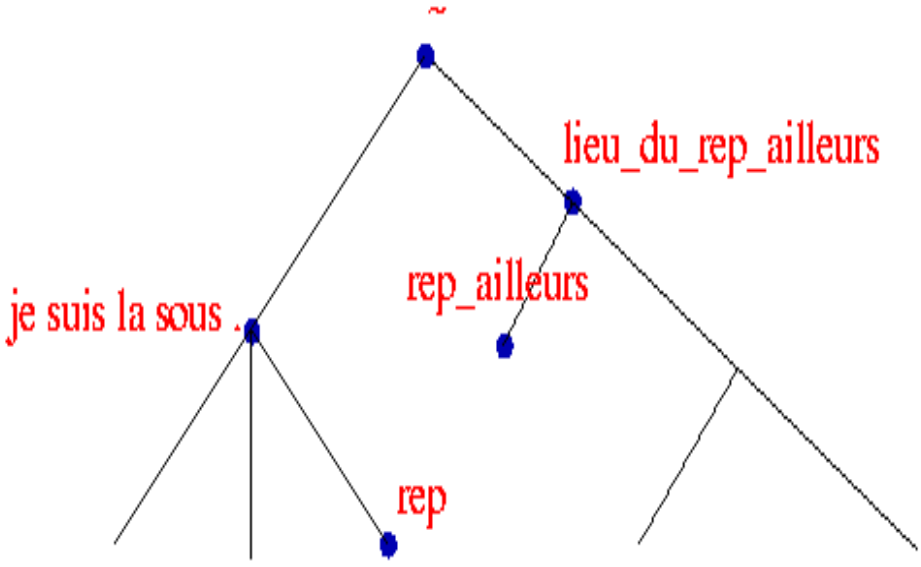
```
Ma_machine>cd bin
Ma_machine>cd bin/new
Ma_machine>cd .././src
```

Sur certains environnements, une variable d'environnement *$HOME* permet de définir de manière générique son HOME. Plusieurs autres espaces peuvent être à votre disposition : *$WORKDIR* et *$TMPDIR*
cd $HOME <=> *cd ~* <=> *cd*

A noter deux répertoires spéciaux, le répertoire courant (c'est−à−dire dans lequel vous êtes) représenté par le point **.**, et le répertoire père représenté par le double point **..**
cd . ne sert donc à rien !

cd − permet de sauter d'un répertoire à l'autre. Vous êtes sous *rep1*, faites *cd rep2*, vous êtes maintenant sous *rep2*. Faites *cd −* et vous vous retrouverez sous *rep1*. Maintenant remplacez *rep1* et *rep2* par des noms de répertoire très longs et différents, vous comprendrez tout l'intérêt de *cd −*.

cd peut avoir des comportements étonnants pour celui qui ne connaît pas votre environnement. Vous êtes habitué à faire *cd rep*, avec *rep* un des répertoires de **.**, mais un cd *rep_ailleurs* avec *rep_ailleurs* un répertoire ne se trouvant pas sous **.** mais sous *lieu_du_rep_ailleurs*, c'est plus étonnant. C'est possible grâce à CDPATH, variable d'environnement qu'il suffit de positionner via *CDPATH=***:.:..:***lieu_du_rep_ailleurs* dans votre *.profile*.
Remarque : la liste des répertoires à inclure dans la recherche sont séparés par des **:**



Il est possible d'afficher en permanence la valeur du répertoire courant dans le prompt du shell. La variable d'environnement *PS1* est affichée systématiquement comme prompt et *PWD* contient "en temps réel " la valeur du répertoire courant, en procédant ainsi :

export PS1="$(echo 'machine:$PWD>')"
le prompt Unix sera égal par exemple à :
machine:/usr/local/bin/>

ou

export PS1=$(echo '${PWD##/}"/> "')*
le prompt Unix sera égal alors à :
bin/>

Magique ? Pour mieux comprendre les *$(' $PWD ##*/ "'* lire la suite du cours

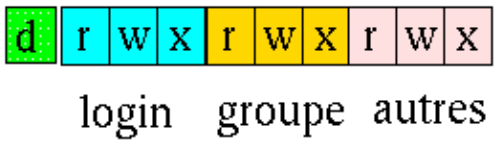
Pour créer un répertoire, utiliser la commande *mkdir mon_nouveau_repertoire*. Si vous voulez créer également tous les répertoires intermédiaires pensez à l'option *−p*

```
Ma_machine>mkdir −p projet/src/new
```

6.6 les droits

Les problèmes d'accès aux fichiers et répertoires sont souvent mal perçus par les débutants. Pourtant les mécanismes de contrôle de lecture/écriture/accès d'un fichier ou répertoire sont simples, mais peuvent devenir un enfer si quelques points d'ombre subsistent.

Les fichiers peuvent être en mode écriture (*w*), en mode lecture (*r*) ou les deux (*rw*), pour vous, pour les membres de votre groupe ou pour les autres (c'est−à−dire pour le reste du monde). Lorsque vous utilisez l'alias créé plus haut (*ll*), vous obtenez en début de chaque ligne "*drwxr−sr−x*" ou "*−rw−r−−r−−*" par exemple.



Le 1er caractère vous indique s'il s'agit d'un fichier (*−*) ou un répertoire (*d*), les trois triplets suivant concernent respectivement le propriétaire du fichier, le groupe du propriétaire, et enfin tous les autres de manière exclusive. Chaque triplet se compose d'un r pour le 1er caractère, d'un w pour le 2e et d'un x pour le 3e, un *−* exprime la négation. Nous avons déjà expliqué les deux premiers caractères *r* et *w*, *x* précise que le fichier peut être exécuté s'il s'agit d'un fichier, ou qu'il peut être "traversé" s'il s'agit d'un répertoire.

Donc si un jour, le shell vous injurie
ksh: mon_code: 0403−006 Execute permission denied.
Les droits d'exécution de votre exécutable ne sont probablement pas bons (x).

Les problèmes peuvent se cumuler. Par exemple, si vous essayez de copier un fichier dans un sous−répertoire (*cp fic1 rep2/rep3/fic1*), il faut que vous puissiez :

1. lire le fichier *fic1* (ce n'est pas forcément évident si le fichier n'est pas le vôtre), droit en lecture sur le fichier
2. traverser le sous−répertoire *rep2* (et tous les autres au−dessus !), *rep2* et *rep3* en *x* au moins pour vous,
3. écrire dans *rep3* (sous *rep2/rep3*, vérifier les droits de .)

Maintenant que tout est clair et que vous avez identifié les éventuels problèmes, la commande *chmod* va permettre de changer les droits des fichiers/répertoires. *chmod* utilise le codage binaire, fondé sur l'association de valeurs numériques aux différentes permissions : lecture : *4*, écriture : *2*, exécution : *1*, pas de permission : *0*.

Chaque triplet se code par l'addition de *4*, *2*, *1*, ou *0*. Pour un *rwx* il faudra ajouter *4+2+1=7*, pour *r−x* *4+0+1=5* etc. Donc les combinaisons vont de *0* (*−* aucun droit) à *7* (*rwx* tous les droits). Cela s'applique à chaque groupe triplet (propriétaire, groupe et autres).

droits	valeur octale	valeur binaire
−−−	0	000
−−x	1	001
−w−	2	010
−wx	3	011
r−−	4	100
r−x	5	101
rw−	6	110
rwx	7	111

La commande *chmod* permettant de positionner *rwxr−x−−−* sur *fic1* à la syntaxe suivante :
chmod 750 fic1

Il existe aussi un autre moyen de positionner les droits d'un fichier avec cette même commande. Vous pouvez utiliser des +, − ou = pour ajouter, supprimer ou fixer des droits à l'une ou l'autre des catégories u (user), g (group), o (other) ou a tous (a) en précisant le type de droit d'accès, r (read), w (write) ou x (execute).

```
chmod g+w fic1
chmod o−x rep
chmod u+rx,g−w fic2
chmod u=rwx,g=rx,o=− fic
chmod −R a+r fic
```

Cette commande est très pratique quand vous souhaitez juste modifier des droits d'une catégorie à un ensemble de fichiers ayant des droits différents les uns des autres.

Si le changement de droits s'applique à un répertoire, vous pouvez changer tous les droits des fichiers et répertoires inclus dans ce répertoire via l'option *−R*.

Si les fichiers que vous créez n'ont jamais les droits que vous souhaitez qu'ils aient, utilisez *umask* (*man umask*)

Bien évidement, il existe dans tous les Window−managers modernes (CDE, KDE etc.) des commandes graphiques permettant de faire la même chose (chmod, mv, cp etc.) mais elles restent dans tous les cas NON standard.

Si les droits sur le groupe sont corrects mais que le nom du groupe est incorrect, vous pouvez via la commande *chgrp* changer le nom du groupe d'un fichier (*chgrp [−R] new_grp fic|rep*, −R pour le faire d'une manière récursive sur un répertoire)
chgrp −R jpp perso

Il existe également une commande pour changer le propriétaire un fichier (mais seul root a ce privilège pour raison évidente de sécurité) *chown [−R] new_owner fic|rep*
chown jpp fic1

Pour un paramétrage encore plus fin, la plupart des implémentations Unix disposent des ACL (Access Control List).
En effet, jusqu'à maintenant nous n'étions pas capables d'autoriser une personne particulière (différente du propriétaire du fichier) à avoir des droits en lecture, écriture etc. sans que toutes les personnes de son groupe aient ces mêmes droits.

Si vous développez un projet avec une autre personne, vous ne souhaitez pas forcément que tout le laboratoire puisse jeter un oeil à votre travail, confidentialité oblige.

Les ACL vont rajouter un niveau de permission sur les droits unix.
Pour faire simple il suffit de protéger le répertoire du projet et laisser les permissions unix classiques sur les fichiers contenus dans le répertoire.

```
Ma_machine$>mkdir projet_proteger_acl
Ma_machine$>setfacl −m  user:login_de_mon_copain:7,m:7 projet_proteger_acl
```

Pour vérifier la bonne affectation de ces nouveaux droits, utilisez *getfacl* dès que vous voyez un + dans le *ls*.

```
Ma_machine$>ls −lda testacl
drwx------+  2 jpp      staff        512 Mar 12 17:22 testacl


Ma_machine$>getfacl projet_proteger_acl
# file: projet_proteger_acl
# owner: jpp
# group: staff
user::rwx
user:login_de_mon_copain:rwx   #effective:rwx
group::---                     #effective:---
mask:rwx
other:---
```

Pour supprimer un droit ACL, seule la commande *setfacl −d user:login* peut être utilisée.

Le traitement qui a été effectué sur *user* peut également être fait sur *group*. Pour plus d'informations *man setfacl*.

Attention sur linux, les ACL ne sont pas installées en standard et doivent être rajoutées par des paquets supplémentaires.

7 A la recherche des fichiers

L'arborescence des fichiers sous Unix n'est pas si compliquée que ça. En fait cela dépend de votre degré d'organisation ou de celle de votre ingénieur système. Unix a un lourd passé, et garde des traces de son histoire au sein de son arborescence, mais essayons de clarifier tous ces répertoires.

/	là où tous les autres répertoires sont montés (accrochés)
/bin	une partie des binaires du système et quelques commandes (ls, cat, rm ..)
/home	partie où sont stockés les fichiers propres aux utilisateurs
/etc	quelques fichiers de configuration et des fichiers systèmes pour le démarrage
/var	fichiers temporaires de quelques démons, de spools d'email et d'imprimantes, de logs, de locks ...
/opt	lieu d'installation préféré des logiciels "modernes"
/boot	image du noyau pour Linux
/dev	ensemble des devices (clavier, disques, cartes (son et réseau) etc.)
/usr	espace "standard"
/usr/bin	pour les binaires
/usr/lib	pour les bibliothèques
/usr/include	pour les "includes" (les .h)
/usr/local/bin	espace "non standard", rajout en local
/usr/local/lib	idem pour les bibliothèques
/usr/local/include	pour les "includes"
/usr/local/src	pour les sources

Lors des installations, il vaut mieux que les fichiers binaires soient avec les binaires, les includes avec les includes et les lib avec les lib ! C'est simple à dire, mais pas forcément facile à faire, certains produits s'installent complètement sous /usr/src ou sous /opt et on se retrouve vite avec des bibliothèques et des includes dans tous les sens, ce qui est quasi impossible à gérer.

Si vous installez un produit chez vous sur votre Home, vous pouvez vous aussi créer un ~/bin, ~/lib et ~/include.

7.1 find

Dans le cas où des fichiers ne sont pas aux "bons" endroits, il faut être capable de les retrouver. Pour cela une seule commande, *find*. Elle paraît toujours un peu compliquée au début, mais on lui pardonne très vite, lorsqu'on découvre sa puissance !

Voilà des exemples qui résument les possibilités de *find* (*find "à partir de" "que faire" "que faire" etc.*) :

- cet exemple va chercher tous les fichiers dont le nom comporte la chaîne de caractères *exe1* dans le répertoire *rep1* ainsi que tous ses sous−répertoires. Attention, la recherche est récursive.

```
Ma_machine$>find rep1 -name "*exe1*" -print
/home/login1/rep1/exe1
Ma_machine$>
```

- Si vous cherchez un include particulier (ici *signal.h*) sans avoir la moindre idée de sa localisation :
  find / −name "signal.h" −print
  Attention, cela risque de prendre un peu de temps surtout si vous avez des disques de plusieurs giga octets ! De plus si vous n'êtes pas root, un grand nombre de messages vont apparaître indiquant l'impossibilité de traverser certains répertoires.

- Si vous voulez tous les fichiers *core* égarés sur votre disque et qui consomment de la place inutilement :
  find ~ −name "core" −print

- Encore plus fort, si vous voulez supprimer directement les fichiers trouvés, il est possible plutôt que de faire afficher seulement les noms de ces fichiers, de les détruire également :
  find ~ −name "core" −exec rm {} \;
  Remarque : *{}* permet de passer les noms des fichiers trouvés à la commande *rm* et le *\;* à la fin est obligatoire.

- Avec deux filtres c'est possible !
  find ~ \(−name ".L" −o −name "*.o" \) −exec rm {} \;*

  find ~ −name ".[Lo]" −exec rm {} \;*

- En fait tout est possible, les filtres sont évalués de gauche à droite. Ici on essaye de supprimer les fichiers contenant "chaine" ; si on n'arrive pas à faire le rm, on affiche le fichier
  find . −exec grep −q chaine {} \; ! −exec rm {} \; −print

- Il est possible de spécifier dans le filtre le type d'élément cherché (fichier *−type f*, ou répertoire *−type d*) ou ayant une taille particulière (*−size nk*), le *−ok* a le même rôle que *−exec* avec la confirmation en plus.
  find . −type −d −print
  find . −size 0k −ok rm {} \;

- Il est également possible de spécifier la date de l'élément cherché (*−atime +n* avec n le nombre de jours depuis le dernier accès)
  find / \(−name "a.out" −atime +7\) −o \(−name ".o" −atime +30\) −exec rm {} \;*

- Un autre cas pratique, vous n'arrivez pas à faire un *rm* sur un fichier car celui−ci contient un caractère "bizarre". Nous allons demander à *find* de détruire (avec confirmation via un *−ok*) un fichier par son numéro d'inode.
  ls −i (pour trouver son numéro d'inode ici 733)

find . −inum 733 −ok rm {} \;

- Remarque sur le *rm* : si vous voulez supprimer un fichier qui commence par
  - ◊ **−** , vous allez avoir quelques surprises, alors utilisez l'option **−−** de *rm* ainsi :
    rm −− fic_avec_un−
  - ◊ **#** , faites *rm ./#fic_avec_un#* ou *rm \#fic_avec_un#*

7.2 PATH

Maintenant que nous avons retrouvé le fichier que nous cherchions (sous */home/login1/rep1/exe1*), nous aimerions bien qu'il soit accessible simplement sans le chemin complet ! Pour cela, il suffit de rajouter ce chemin dans la variable *PATH*, dans votre *.profile* sans écraser l'ancienne valeur de PATH :
export PATH=$PATH:/home/login1/rep1/

Ainsi votre exécutable *exe1* pourra être exécuté de n'importe où en tapant "*exe1*". Attention à ne jamais écraser le *PATH* initial car il contient déjà l'essentiel des répertoires utiles (si vous l'écrasez, il n'y aura pas grand chose qui marchera !).

Maintenant vous êtes en mesure de comprendre pourquoi une commande présente dans votre répertoire courant ne s'exécutait que par *./ma_commande* et non simplement par *ma_commande*, le . (c'est−à−dire le répertoire courant) n'était pas dans votre *PATH* ! Le shell était alors incapable de la trouver, même si elle se trouvait sous "ses" yeux !

Attention à l'ordre car c'est le premier exécutable trouvé qui sera exécuté.
Faites toujours plutôt *PATH=$PATH:.* que *PATH=.:$PATH*, car dans ce deuxième cas une commande peut avoir une signification différente suivant le répertoire où elle est exécutée.

7.3 type

Parfois c'est le cas contraire, le shell "trouve" la commande, c'est−à−dire l'exécute sans problème, mais l'utilisateur, ignore où elle se trouve ; bien difficile alors de la modifier. En effet la variable PATH peut contenir un grand nombre de répertoires différents et rendre délicate la recherche de l'exécutable. Pour connaître le chemin exact de la commande exécutée, utilisez *type* :
type ma_commande
Elle précisera le chemin absolu utilisé pour exécuter *ma_commande*.

```
Ma_machine>type super_menage.ksh
super_menage.ksh is /home/sos/jpp/bin/super_menage.ksh
```

Cas réel : j'ai modifié un script *super_menage.ksh* qui se trouve directement dans mon *HOME*, et les modifications semblaient ne pas être prises en compte bien que je la lance depuis mon HOME !

Puisque *type* me donne */home/sos/jpp/bin/super_menage.ksh*, cela veut dire que si j'exécute *super_menage.ksh*, le script exécuté sera celui sous *bin*, et non celui du *HOME* ! Pourquoi ? Une seule explication : *bin* est placé avant le "." dans mon *PATH*.

7.4 diff

Bref, nous avons donc retrouvé tous les fichiers/commandes/scripts "perdus", mais malheureusement il arrive parfois que l'on ait 2 ou 3 fichiers quasiment identiques, des versions différentes d'un même code/script (*~super_menage.ksh* et *~/bin/super_menage.ksh* par exemple), ou bien des sorties d'un même travail, sans savoir vraiment ce qui les diffère. Imaginons que ces fichiers fassent plusieurs centaines de lignes, comment faire pour examiner leurs contenus ? Pour quelques centaines/milliers de lignes aucune solution manuelle... La solution rapide et efficace est *diff*, cette commande extrait uniquement les différences entre deux fichiers. Si 2 lignes diffèrent sur 10000 lignes, seules ces deux lignes apparaîtront à l'écran !

Dans cet exemple qu'est−ce qui diffère entre mes deux scripts *super_menage.ksh* ?

```
Ma_machine>diff ~/super_menage.ksh ~/bin/super_menage.ksh
>if (( $? ==0))
> then
> rm poub tempo
>fi
```

Les < (respectivement les >) indiquent que seuls ces éléments se trouvent dans le premier (respectivement dans le deuxième) fichier.

Remarque : *diff* a une option *−r* lui permettant d'explorer des répertoires et d'en déterminer les différences, en comparant tous les fichiers deux à deux dans tous les sous−répertoires.

8 Comment communiquer son travail à quelqu'un d'autre ?

Après quelques semaines, vous désirez transférer votre travail à un membre de votre équipe. Mais voilà, tous les fichiers sont éparpillés sur votre disque...

8.1 tar

Le premier réflexe est de les regrouper dans un seul répertoire (via un *cp*) puis de "tarer" ce répertoire, c'est−à−dire, de rassembler toute l'arborescence du répertoire en un seul fichier *mon_boulot.tar*. Si vos fichiers sont sous *rep1* dans votre *HOME*, alors vous pouvez utiliser sous votre *HOME*, la commande *tar* ainsi :
tar −cvf mon_boulot.tar rep1
le fichier *mon_boulot.tar* sera ainsi créé.

Pour détarer le c (create) devient x (extract) ainsi :
tar −xvf mon_boulot.tar
l'arborescence de *rep1* sera complètement reconstituée sous le répertoire courant.

Si vous souhaitez connaître le contenu du *tar* sans le "détarer", utilisez l'option *t* ainsi :
tar −tvf mon_boulot.tar
Cette option est très utile ...

- Imaginez que vous "détariez" sur votre Home un fichier tar contenant 200 fichiers . Sans arborescence, ces 200 fichiers iraient se mêler à vos fichiers du Home, bon courage pour le nettoyage si vous souhaitez supprimer tous les fichiers.
- Plus grave, si un fichier issu du tar a le même nom qu'un fichier déjà existant sur votre Home ... votre fichier sera remplacé !

Une fois votre fichier repéré, pour l'extraire de l'archive (et seulement lui) rajoutez−le à la commande habituelle :
tar −xvf mon_boulot.tar mon_fichier

Remarque : si vous souhaitez rajouter un fichier supplémentaire à votre fichier tar, vous pouvez le faire grâce à l'option *−r* ainsi *tar −rvf mon_boulot.tar fichier_suppl*

Ceci dit, si votre code tient dans un seul fichier, rien ne vous empêche de l'envoyer par mail puisqu'il est en format ASCII, par la simple commande Unix :
mail mon_copain@autre_labo.fr < super_menage.ksh (l'utilisation de < sera vue plus loin)
Attention, si vous utilisez la possibilité "d'attacher" un document via votre *mailer* favori, assurez vous que :

1. votre *mailer* soit bien configuré, (envoyez−vous un mail avec "attachement" pour vérifier)
2. votre destinataire dispose d'un *mailer* acceptant un attachement ! (c.−à−d. qu'il soit de type MIME, c'est le cas avec netscape, exmh, mutt, IE etc.)

Faites aussi attention à l'espace d'arrivée de votre courrier, veillez à bien configurer le `.forward`. Ce fichier sert à renvoyer vos messages vers une autre machine, cela permet de n'avoir qu'une boîte aux lettres à vérifier. Il doit contenir une de ce type
autre_login@un_autre_labo.ailleur.fr

8.2 gzip

Le fichier créé par un tar est souvent volumineux, il est alors judicieux de le compresser par de puissants outils afin de gagner de la place disque (60−70%) et du temps lors du transfert du fichier. Les deux compresseurs les plus souvent employés sont *compress* (standard Unix) et *gzip* (produit domaine public du GNU). Ce dernier a de grandes chances d'être déjà installé sur votre machine (automatiquement installé sous Linux), de plus il est en général plus efficace que *compress*. La compression/décompression s'effectue ainsi :

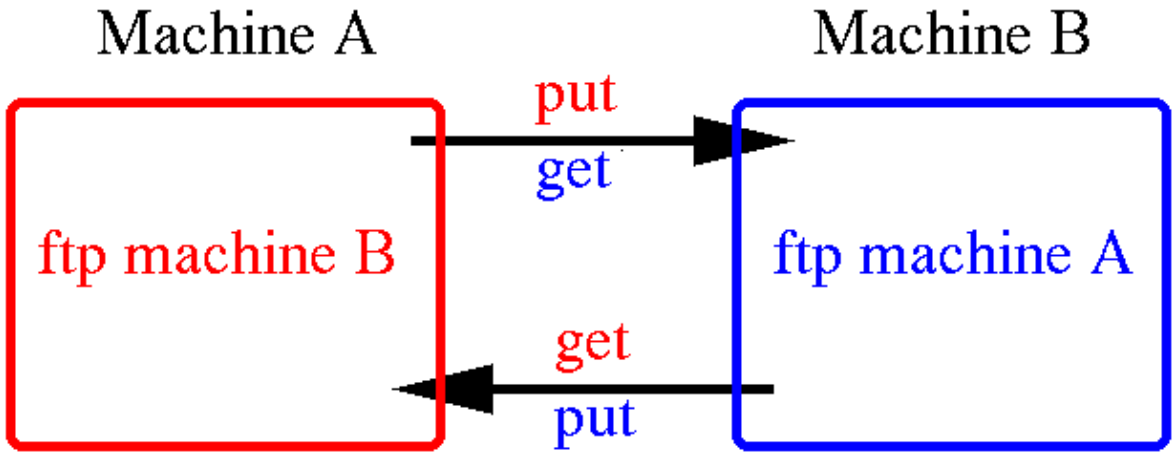
commandes	compression	fichier compressé généré	décompression
compress	*compress mon_boulot.tar*	*mon_boulot.tar.Z*	*uncompress mon_boulot.tar.Z*
gzip	*gzip mon_boulot.tar*	*mon_boulot.tar.gz*	*gunzip mon_boulot.tar.gz*

Remarque : il est inutile de compresser un fichier déjà compressé ! On ne gagnera rien, pire il sera probablement même plus gros !

8.3 ftp

Maintenant, il ne reste plus qu'à transférer le fichier "tar" sur la station du destinataire via ftp pour cela vous devez juste,

- soit connaître son login/passwd (pas recommandé),
- soit avoir votre propre login sur cette machine.



La commande s'utilise ainsi :
ftp autre_machine.autre_domaine.fr
elle vous invite à rentrer le *login* et le *passwd* du compte distant,
puis tapez :
put mon_boulot.tar.gz

Le fichier sera alors placé dans le HOME du login distant.
Remarque : si vous avez plusieurs fichiers à transférer, vous pouvez utiliser "*mput *.gz*". Il est possible de

s'affranchir des YES (return) pour chaque transfert en désactivant le prompt via la commande *prompt* avant le *mput* ou *mget*.

Sous *ftp*, les commandes *ls, pwd* et *cd* peuvent être utilisées pour lister et changer de répertoire sur la machine distante, et *!ls*, *!pwd* et *!cd* pour faire de même sur la machine locale .

Dans le cas où vous avez laissé le fichier chez vous, et avez simplement dit à votre collègue de venir le prendre via un *get* sous un *ftp*, pensez à mettre les droits de lecture sur ce fichier et lui laisser la possibilité de traverser votre *HOME*.
Reste ensuite au login distant à décompresser (on l'a vu !) et à détarer via :
tar −xvf mon_boulot.tar

Remarque sur le mode binary ou ascii de ftp.
Si vous restez dans le monde Unix(/linux), vous n'avez pas à vous soucier du mode quels que soient les machines utilisées et les fichiers transférés.

Attention, si vous transférez à partir de ou vers un PC sous Windows ! Si c'est un fichier texte, précisez le mode ascii. D'autres environnements comme IRIX préfèrent que vous stipuliez `bin` ou `ascii`.

Pour sortir de la commande *ftp*, tapez *quit*.

8.4 Rcommandes

Comme nous en sommes aux interactions entre deux machines, détaillons un peu les Rcommandes. Il s'agit des *rsh*, *rcp*, *rlogin*...
Toutes ces commandes seraient déjà connues si elles n'avaient pas de r (*login*, *cp* ...). Le r (pour remote) précise que la commande va s'exécuter sur une autre machine.

Pour que ces commandes fonctionnent, il est **impératif** d'avoir son fichier .rhosts à jour. Ce fichier permet de s'affranchir du mot de passe normalement nécessaire lors d'une connexion sur une autre machine.

Voici un exemple de fichier *.rhosts* existant sur le *HOME* de la *machine_A* pour le compte *login_A* .

```
machine_B.mon_labo.fr login_B
eclipse.totale.fr observateur
```

Ici, sur la *machine_A* peuvent se connecter sans taper de mot de passe, *login_B* depuis *machine_B* et *observateur* depuis *eclipse*.

- *rcp [−r] [[loginA@]machineA:]/fichier [loginB@[machineB:]]/fichier* : copie d'une machine sur une autre, loginX et machineX peuvent être omis. L'option *−r* permet de traiter une copie récursive pour recopier les répertoires.

  ```
  rcp −r observateur@eclipse.totale.fr/.netscape .
  ```

- *rsh machineB commande* : exécute `commande` sur la `machineB` (alors que rsh a été exécutée sur la machine courante, dans l'exemple suivant le fichier *liste_des_utilisateurs_distants* est créé sur la

machine courante !)

```
rsh eclipse.totale.fr who > liste_des_utilisateurs_distants
```

- *rlogin machineB [−l login]* : établit une connexion sur la `machineB` (comme la commande *telnet* mais sur un autre port) sous le login indiqué

```
rlogin machine −l jpp
... connexion direct ...
```

Attention il est parfois nécessaire de préciser le path de la commande distante.

Remarque : il est fortement conseillé de vérifier le contenu du fichier *.rhosts* régulièrement. Supprimer toutes les entrées inutiles, ou les éventuels rajouts de personnes mal intentionnées.

9 L'éditeur et le contrôle de la ligne de commande

9.1 vi/emacs/nedit

Le monde de l'édition est un monde encore plus conflictuel que celui des shells. Il existe de multiples éditeurs sous Unix, mais deux majeurs s'affrontent depuis 1975 ! *Vi* d'un côté (standard Unix) et *Emacs* (domaine public, à installer) de l'autre.

Chacun a ses propres avantages et défauts, le seul conseil serait : "prenez le même que votre collègue de bureau !", il pourra ainsi vous renseigner et vous aider (aucun des deux n'est vraiment simple au début). Pour ma part, je suis convaincu qu'emacs est mieux, mais son côté "usine à gaz" fait que j'utilise toujours vi ! (enfin vim, version améliorée de vi) (et mon collègue de bureau aussi !).

Ceci dit, *nedit* est également disponible sur la plupart des machines et ne pose aucun problème au débutant (sauf pour le trouver, il est sous */usr/local/pub/bin*). Il s'utilise comme un traitement de texte et colorie les mots clés, par contre il n'offre pas toutes les possibilités de *emacs* ou *vi*.

vi utilise des fichiers temporaires, et parfois l'espace où il stocke ses fichiers est plein. Pour utiliser un autre espace, positionner ainsi EXINIT, *export EXINIT="set dir=autre_file_system"*.

Je vous conseille quand même *vi*, car on retrouve les mêmes séquences dans les expressions régulières, les crons, sed, ed etc.

Les principales commandes sous *vi*.

Commandes	Fonctions
i	basculement en mode insertion
a	basculement en mode ajout
cw	modification du mot courant
esc	sortie du mode d'insertion/ajout/modification
J	concaténation de la ligne courante et de la suivante
x	effacement du caractère suivant
X	effacement du caractère précédent
$	déplacement en fin de ligne
0	déplacement en début de ligne
dd	suppression de ligne et copie dans le buffer
dw	suppression de mot et copie dans le buffer
p	copie du buffer sous la ligne courante
D	effacement jusqu'à la fin de la ligne
.	répétition de la dernière commande
u	annulation de la dernière commande
h	déplacement vers la gauche

l	déplacement vers la droite
j	déplacement vers le bas
k	déplacement vers le haut
CTRL F	déplacement sur la page suivante
CTRL B	déplacement sur la page précédente
:	entrée dans le mode de commande pour les commandes suivantes
w [fic]	sauvegarde dans le/un fichier
q	sortie de vi
q!	sortie sans sauvegarde
x	sortie avec sauvegarde
r fic	insertion de fic dans le fichier courant
! cmd_du_shell	exécution d'une commande shell
%s/chaine1/chaine2/g	substitution dans tout le fichier de chaine1 ou rexp par chaine2 (plusieurs fois par ligne)
sx,y/chaine1/chaine2/	substitution de x à y de chaine1 ou rexp par chaine2
s4,./chaine1/chaine2/g	substitution de la 4e ligne à la ligne courante de chaine1 ou rexp par chaine2 (plusieurs fois par ligne)
s4,$/chaine1/chaine2/gc	substitution de la 4e ligne à la fin de chaine1 ou rexp par chaine2 (plusieurs fois par ligne) avec confirmation (y,n,q)
g/chaine/l	liste des lignes contenant chaine

Il est possible pour certaines commandes d'utiliser un facteur multiplicatif devant comme *7x* ou *3dd*.

Une bonne gestion et édition des lignes de commande est fondamentale sous Unix mais dépend fortement de votre shell. Il y a quatre choses à savoir faire absolument (car on le fait 200 fois par jour) :

Shell	rappel de la dernière commande	rappel d'une commande cmd	déplacement dans la ligne de commande	complétion
ksh (set −o vi)	*Esc k,* puis *n* ou *N* pour naviguer dans l'historique commande suivante/commande précédente	*Esc/* suivie d'une sous chaîne, par exemple *Esc/totoCR* de retrouver toutes les commandes où le toto était présent	puis *l* ou *h* pour aller de droite à gauche puis *x* pour supprimer et *a* pour ajouter	*titiEsc* ou *titiEsc=* listera tous les fichiers commençant par *titi*
ksh (set −o emacs)	*Ctrl p* *Ctrl n*	*Ctrl r cmd*	*Ctrl b* *Ctrl f*	*EscEsc*
ksh (set −o	flèche "haut/bas"	*Ctrl r cmd*	flèche "droite/gauche"	*EscEsc*

emacs) + config flèche				
csh	**!!**	*! debut_cmd*		
bash (linux)	flèche "haut/bas" ou **!!** ou *Ctrl p* *Ctrl n*	*Ctrl r cmd* ou *! debut_cmd*	flèche "droite/gauche" ou *Ctrl b* *Ctrl f*	*TAB* ou *EscEsc*

Config flèche : elle n'est possible qu'en mode emacs. Il faut placer les commandes suivantes dans l'un de vos fichiers d'environnement (.profile par exemple) :

```
alias __A='^P' # pour remonter dans l'historique des commandes (flêche ascendante)
alias __B='^N' # pour descendre dans l'historique des commandes (flêche descendante)
alias __C='^F' # pour se déplacer à droite sur la ligne de commande (flêche droite)
alias __D='^B' # pour se déplacer à gauche sur la ligne de commande (flêche gauche)
```

Attention : le caractère "^P" représente un seul caractère : le caractère "Ctrl−p" et non pas "^" puis "P". Pour saisir ce caractère "Ctrl−p" sous l'éditeur emacs il faut taper la séquence de touches "Ctrl−q Ctrl−p"

Pour passer en mode *vi* ou *emacs* sous *ksh*, il faut insérer dans votre *.profile* cette ligne *set −o vi* ou *set −o emacs*

Pour la gestion de l'historique, deux variables d'environnement existent et peuvent être placées dans votre .kshrc. Une pour la taille de l'historique `HISTSIZE` et une pour nommer l'historique `HISTFILE`.

```
HISTSIZE=5000
HISTFILE=~/.sh_history
```

Tous les shells acceptent des **expressions régulières** comme :

- * : équivaux à "n'importe quoi" (chaîne, caractère ...) sauf . et /
  *ls *.c* listera les fichiers ayant comme suffixe un c
- ? : remplace un caractère dans une chaîne
  cat t?t? affichera le contenu de toto, titi, tata et t1t2
- [group]: sélectionne certains caractères
  cat t[ia]t[ia] affichera le contenu de titi et tata, mais pas toto ni t1t2
- [début−fin]: sélectionne un intervalle
  cat [0−9][!0−9]* affichera le contenu de tous les fichiers ayant un chiffre au début mais sans chiffre à la fin (dû au !).

Attention aux caractères spéciaux, si vous souhaitez qu'ils ne soient pas interprétés comme le $ () [] " (double quote) # (dièse) ' (quote) ` (back quote) \ (back slash): utilisez le \ (back slash) juste avant.

Exemple : *echo il ne faut pas confondre / et \> en Unix.*

Trois quotes à ne pas confondre :

- si vous voulez que rien ne soit interprété, placez des ' (quotes) de part et d'autre de votre chaîne.
  *>echo '$?*_\'*
  *>$?*_*
- si vous voulez quand même interpréter les dollars, utilisez les " (doubles quotes),
  *>echo "$PATH?*_\"*
  *>/usr/bin?*_*
- si vous voulez exécuter une chaîne, il est nécessaire de la placer entre ` (back quotes).
  >REP=`ls −lrt *.rpm`;echo $REP
  >−rw−r−−r−− 1 jpp jpp 134226 nov 21 17:51 libcdf−2.7−1.i386.rpm

9.2 ed/sed

Ils sont utilisés comme éditeurs non interactifs à base de commande *vi*. *sed* lit les lignes d'un fichier une à une sans modifier le fichier source et applique le traitement souhaité. *ed* quant à lui, travaille directement sur le fichier source.

Quelques exemples :

- substitution des occurrences d'une chaîne (faux) par une autre (vrai) :
  sed 's/faux/vrai/' fic1 > fic2 remplacera la 1ère occurrence de la chaîne, pour chaque ligne du fichier.
  sed 's/faux/vrai/3' fic1 > fic2 remplacera la 3e occurrence de la chaîne.
  sed 's/faux/vrai/g' fic1 > fic2 remplacera toutes les occurrences de faux par vrai.
  sed 's/[Ff]aux/vrai/g' fic1 > fic2 remplacera toutes les occurrences de Faux et faux par vrai.

- suppression de lignes :
  sed '20,$d' fic1 supprimera les lignes de la 20e à la dernière.
  sed '/commentaire/d' fic1 supprimera toutes les lignes ayant le mot "commentaire".
  printf '1d\nwq\n' | ed fic1 supprimera la première ligne du fichier fic1.
  printf 'g/^$/d\nwq\n' | ed fic1 supprimera toutes les lignes blanches.

Lorsque le code *sed* est trop long, vous pouvez mettre les commandes dans un fichier et appeler *sed* ainsi :
sed −f fic_commande_sed fic1 > fic2

10 La gestion des flux et des processus

Un des principaux problèmes pour les débutants sous Unix, c'est d'arriver à gérer correctement trois symboles :
>(redirection de sortie), < (redirection d'entrée), | (pipe).

On peut voir ces symboles comme une écriture sur un fichier pour le supérieur, une lecture à partir d'un fichier pour l'inférieur, un simple tuyau pour le pipe. Ce tuyau servant à relier deux commandes entre elles, c'est−à−dire, la sortie de l'une dans l'entrée de l'autre.

Nous allons voir les commandes Unix comme un jeu de construction.

10.1 Fonctions et Connecteurs

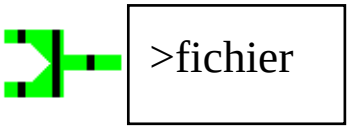
- entrée commande (0)

- sortie commande (1)

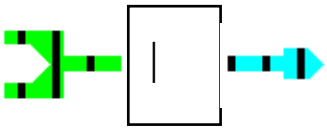
- sortie erreur (2)

- commande type

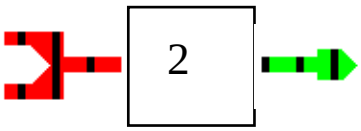
- création d'un fichier contenant le flux de la sortie commande



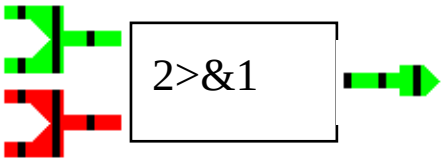
- connection de deux commandes



- contrôle de l'erreur

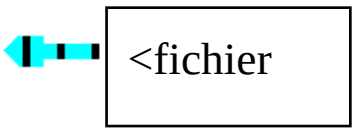


- fusionne l'erreur et la sortie commande



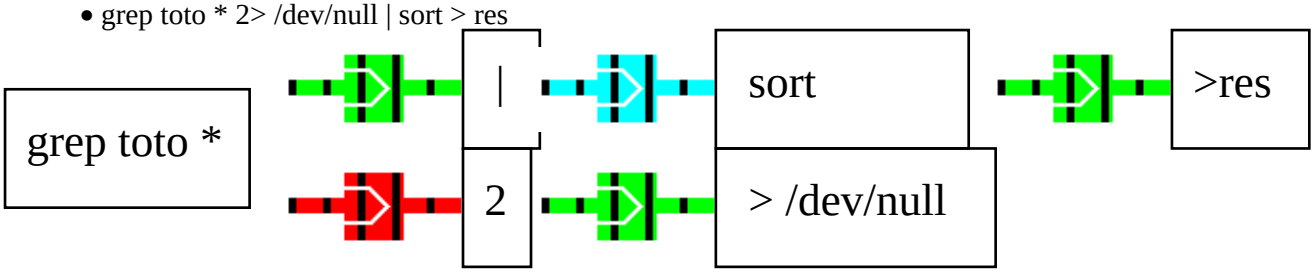
- alimente la commande via l'entrée commande

- alimente la commande via la lecture d'un fichier



Deux exemples avec ces symboles

- grep toto * 2> /dev/null | sort > res



- mon_script 2>&1 | mail jpp@labo

Maintenant que vous avez compris le principe voici une série d'exemples :

- Exemple d'écriture sur un fichier avec > :
  - ♦ Les informations données par *date* vont être placées dans *mon_premier_fichier*.
    date > mon_premier_fichier
    Attention, si le fichier existe il sera détruit ! Sauf si vous avez positionné `set -o noclobber`. Dans ce cas, si vous voulez vraiment l'écraser il vous faudra faire :
    date >! mon_premier_fichier

  - ♦ Les informations retournées par *who* vont être ajoutées à *mon_premier_fichier*.
    who >> mon_premier_fichier

    $>cat mon_premier_fichier
    Thu Oct 29 15:39:56 MET 1998
    sfuj742 pts/0 Oct 28 08:00 (xt2−fuji.idris.f)
    jpp pts/5 Oct 29 08:32 (buddy.idris.fr)
    lavallee pts/8 Oct 29 08:39 (bebel.idris.fr)
    gondet pts/9 Oct 29 08:48 (glozel.idris.fr)
    $>

- Exemple de lecture d'un fichier avec < :

- Si vous souhaitez envoyer *mon_premier_fichier* par mail à *assist@microsoft.fr*, il faut que la commande *mail* lise votre fichier, pour cela nous allons utiliser "<" (la redirection d'entrée) ainsi :
  mail assist@microsoft.fr < mon_premier_fichier

- Si vous avez un script *mon_script* qui vous pose de multiples questions, vous pouvez l'automatiser grâce au <. Mettez les réponses dans le fichier *les_reponses* et faites comme suit :
  mon_script < les_reponses

- Avec les deux symboles :
  mon_script < les_reponses > les_sorties

- Avec deux < on envoie à la commande toutes les lignes tapées en entrée, jusqu'à la chaîne `fin`, méthode du "here document" :
  wc −l << fin
  "collez quelque chose avec la souris"
  fin

- Encore avec deux < on envoie à la commande toutes les lignes tapées en entrée, jusqu'à la chaîne `fin` mais cette fois−ci dans un fichier :
  cat > fic << fin
  "collez quelque chose avec la souris ou tapez du code ..."
  fin

- Exemple d'utilisation d'un pipe | :
  - Voilà une autre solution pour envoyer un mail :
    cat mon_premier_fichier | mail assist@microsoft.fr

  - Si l'on cherche à connaître le nombre de fichiers d'un répertoire :
    ls −la | wc −l

  - Si une commande est trop bavarde et génère plusieurs écrans, canalisez−la par :
    commande_bavarde | more

  - Le pipe sert très souvent à remplacer le fichier input attendu dans la plupart des commandes par un flux de la commande précédente, *cat * | sort*
    à la place de
    *sort **

Attention : le flux redirigé dans un > ou un | ne concerne que la sortie standard, pas l'erreur standard !

- Pour fusionner les deux flux (sortie et erreur), utilisez cette syntaxe : *commande 1> poub 2>&1* qui redirigera le flux 2 vers le flux 1 (attention uniquement avec la famille sh).

- Pour rediriger la sortie d'erreur dans un fichier (sans bloquer le flux standard) :
  commande 2>erreur

- Si vous voulez que les sorties ne soient pas affichées ou conservées dans un fichier, il suffit de les envoyer vers */dev/null* ainsi :
  script_bavard > /dev/null

- Pour rediriger la sortie d'erreur dans un pipe sans le flux standard :
  (commande 1>/dev/null) 2>&1 | more

10.2 xargs/sh

Pour exécuter un flux, la commande *xargs* peut vous aider :
`-t` pour avoir l'echo des commandes et `-n` pour découper le flux d'entrée en paquets, et `-i` pour insérer le flux.

- ```
  $>xargs -t -n 2 diff  <<fin
  fic1 fic2 fic3
  fic4 fic5 fic6
  fin
  diff fic1 fic2
  diff fic3 fic4
  diff fic5 fic6
  ```

- ```
  $>ls | xargs -t -i{} mv {} {}.old
  mv chap1 chap1.old
  mv chap2 chap2.old
  mv chap3 chap3.old
  ```

- ou dans un cas plus simple avec *sh*
  ```
  $>echo "ls; pwd" | sh
  chap1.old chap2.old chap3.old
  /home/jpp/rap
  ```

10.3 grep

`grep` permet de rechercher les occurrences d'un mot ou d'un morceau de mot dans un fichier. Si vous voulez trouver toutes les personnes qui utilisent le *ksh* (bash sous linux) pour leur demander un petit conseil, sachez que le fichier */etc/passwd* contient la liste des logins de votre machine avec, en particulier, le shell qu'ils utilisent. Il suffit de faire alors :

```
$>grep bash /etc/passwd
teuler:*:505:502:teuler:/home/teuler:/bin/bash
lavallee:*:506:502:lavallee:/home/lavallee:/bin/bash
corde:*:507:502:corde:/home/corde:/bin/bash
```

---

grep accepte des expressions régulières (comme *chaine, ^chaine, *toto*, chaine$) et accepte aussi de multiples options. On peut voir les expressions régulières comme des filtres, des masques agissant sur des chaînes de caractères. Les cinq options du *grep* les plus utiles sont :

- *-i* pour ne pas tenir compte des majuscules/minuscules,
- *-v* pour ignorer un mot,
- *-n* pour avoir les numéros de ligne,
- *-E* pour les expressions régulières plus compliquées,

- *−l* pour lister seulement les fichiers contenant la chaîne recherchée,
- *−s* pour supprimer les messages d'erreurs.

Pour comprendre voici quelques exemples :

- nombre de lignes en commentaire (commençant par !) dans un code Fortran :
  grep "^!" prog.f | wc −l

- recherche de STOP avec le numéro de la ligne dans tous les sources :
  *grep −n −i stop *.f**

- liste de tous les fichiers qui n'ont pas "image" ou "son" dans leur nom :
  ls | grep −vE "(image|son)"

- liste des lignes contenant "image" ou "son" dans tous les fichiers du répertoire courant
  *grep −E "(image|son)" **

- liste des fichiers contenant JPP dans une arborescence
  find . −name "" −exec grep JPP {} \;*

Liste des caractères spéciaux	Signification pour les expressions régulières de grep,find,awk ou vi (en partie)	Signification pour le shell
.	caractère quelconque	.
$	fin de ligne	idem
^	début de ligne	idem
[]	un des caractères du crochet	idem
−	de ... à ds [x−y]	idem
?	expression régulière précédente optionnelle	caractère quelconque
*	répétition >=0	chaîne quelconque
+	répétition >0 (pas ds vi)	+
|	ou (pas ds vi)	pipe
()	groupement des expressions (pas ds vi)	groupement des commandes

DÉMO/TD

- Créer un fichier contenant les valeurs de `s1_np` se trouvant dans tous les fichiers du répertoire TP/TP1, combien y en a t−il ?.
- Comptez le nombre d'occurrence de `tp_np` dans les fichiers n'ayant que deux chiffres dans leur suffixe.

10.4 tail

Votre code s'est exécuté et vous souhaitez savoir si tout s'est bien passé. Pour cela il suffit de "voir" la fin du contenu du fichier output (dans le cas d'un job) par la commande *tail* (elle n'affichera que les 10 dernières lignes).

- Si vous souhaitez 25 lignes ajoutez *−n 25* à la commande *tail*
- Si votre code est en train de s'exécuter vous pouvez suivre l'évolution de vos fichiers d'output ou de log par cette même commande agrémentée d'un *−f*.
  tail −f output affichera "en direct live" votre fichier. Sans cette commande, vous seriez obligé d'éditer le fichier et de le recharger toutes les 10 secondes !
- Si vous souhaitez faire le contraire, avoir le début du fichier, utilisez *head output*

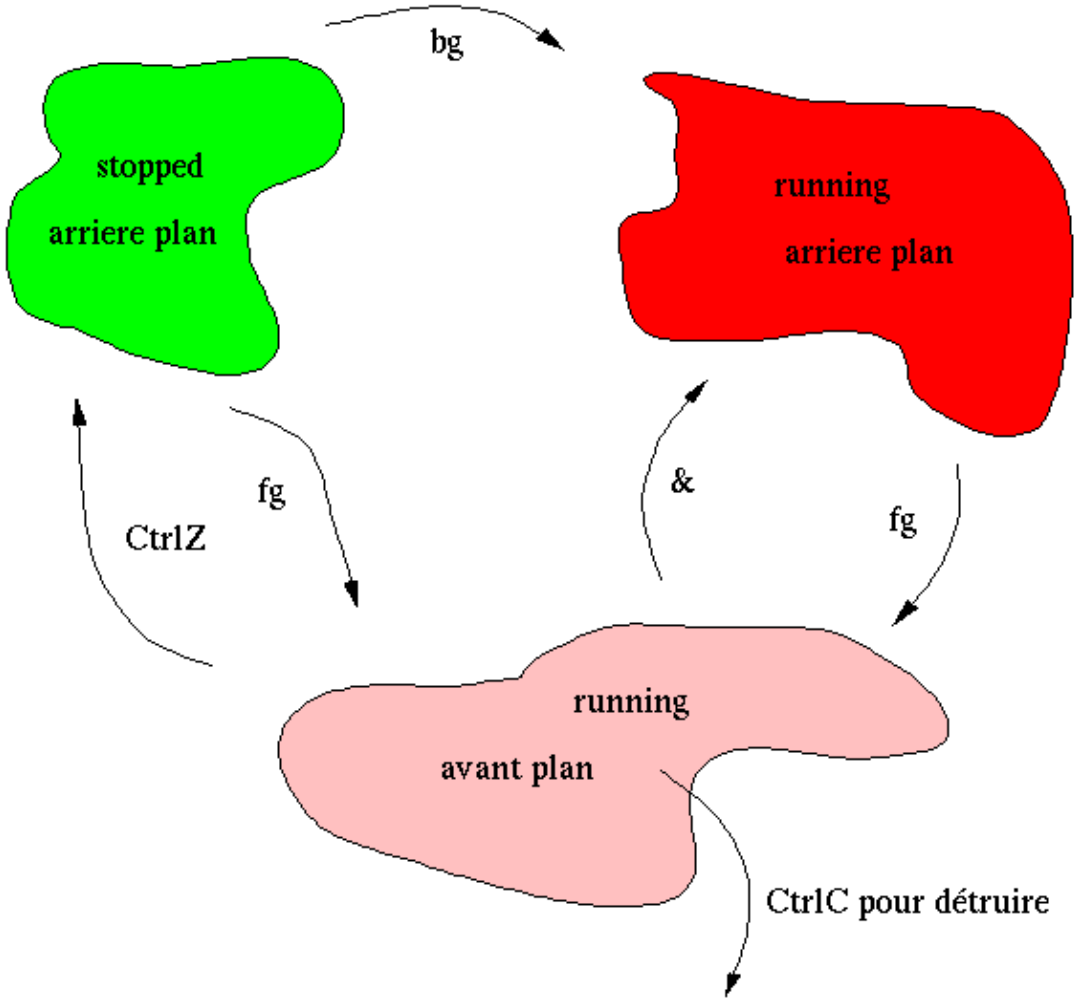
10.5 processus

Lorsque vous utilisez des commandes ou vos programmes favoris sous Unix, ils s'exécutent systématiquement sous votre shell.
Tant que la commande s'exécute, vous n'avez plus la main, vous êtes bloqué. Certains utilisent plusieurs fenêtres pour continuer à travailler.

Quel que soit votre commande ou votre exécutable, c'est en fait un processus Unix qui peut être facilement contrôlé.

Sur une commande un peu longue comme un *find* (que l'on a déjà vu) essayez un *CTRL z*. Cela bloquera le *find* et vous permettra de retrouver la main sous le shell. Votre processus sera suspendu. Pour le réactiver sans être de nouveau bloqué tapez *bg* pour *background* (arrière−plan).

La séquence *CTRLz* suivie de *bg* est identique au & souvent utilisé après une commande. L'avantage de *CTRLz + bg* est qu'il est possible via *fg* (*foreground*, avant−plan) de revenir à la situation de départ (c'est−à−dire comme si vous aviez lancé la commande sans *&* à la fin), un *CRTL c* (permettant de tuer votre commande) est donc toujours potentiellement actif.



Au fait, pourquoi vouloir tuer une commande ? Plusieurs réponses :

- vous vous êtes trompé (ça arrive),
- la commande ne répond pas assez vite ou ne répond pas du tout !
- le résultat produit est suffisant.

Mais comment faire si un *CRTL c* ne marche pas et qu'il n'y a pas de menu "close" dans l'application ? Sous d'autres systèmes, pas grand chose, à part tout redémarrer... Sous Unix, il suffit de trois choses :

- avoir une fenêtre pour taper des commandes, ou pouvoir en créer une (sinon allez voir votre administrateur, *root* (le login qui a tous les droits) a toujours une fenêtre pour les cas d'urgence),
- trouver le numéro de process de votre commande ou application que vous souhaitez tuer. Pour cela (et en général seulement pour ça) faites un "*ps −edf | grep mon_login*" ou "*ps aux | grep mon_login*" suivant votre machine,
- faire un *kill numéro_du_process_à_tuer*. Si ça ne suffit pas, utiliser l'artillerie lourde *kill −9 numéro_du_process_à_tuer*. (Remarque, un kill −1 permet de relancer un process, prise en compte du fichier de configuration pour un démon par exemple.)

```
Ma_machine>ps −edf | grep jpp
     UID    PID   PPID   C    STIME    TTY  TIME CMD
     jpp  16034  57976   0 08:49:52  pts/6  0:00 −ksh
```

```
jpp  18032  83270   0 08:57:38      -  4:35 netscape_aix4
jpp  24616  84396   4 10:12:30  pts/7  0:00 grep jpp
jpp  28788  69868   0 08:49:46      -  0:07 xautolock
jpp  36206  69868   0 08:49:46      -  0:00 xterm
jpp  41836  69868   5 08:49:46      -  0:00 xterm
jpp  44662  69868   0 08:49:46      -  0:00 xbiff
jpp  52346      1   0 08:49:47      -  0:07 plan
jpp  58662  84396  21 10:12:30  pts/7  0:00 ps -edf
jpp  67722  36206   0 08:49:51  pts/5  0:00 -ksh
jpp  67952      1   0 08:49:46      -  0:02 /usr/local/bin/pland -k
jpp  85608  69868   0 08:49:46      -  0:01 xmh
jpp  94304  69868   0 08:49:46      -  0:24 fvwm
```

Prenons un exemple : avec Netscape, je n'ai plus de contrôle, le *File/Exit* ne réagit plus ! Ici Netscape a pour PID (*process IDentification*) *18032*, pour s'en débarrasser :
kill 18032

Cas contraire : je souhaite que ma commande se déroule normalement alors que je voudrais me déconnecter. Si j'exécute une commande et que je sors de ma session (^D ou exit) la commande se terminera aussi ! Pour éviter cela, vous pouvez utiliser la commande *nohup* ainsi :
nohup ma_commande&
L'output de la commande ira dans un fichier nommé *nohup.out* et surtout vous pourrez vous déconnecter ! (plus de raison de laisser une session ouverte après votre départ).

Remarque : démarrage d'une commande à une heure donnée.

- Si vous souhaitez ne lancer qu'une seule fois une commande à une heure particulière sur votre station, c'est possible avec *at*.

```
$>at now +3 minutes
find /home/jpp -name "core" -exec rm {} \;
^D
$>
$>echo who | at now +4 hours
$>
```

Pour plus d'informations : *man at*, la sortie de votre commande à l'exécution de *at* vous sera envoyée par mail !

- Si vous souhaitez que l'exécution soit périodique et automatique (dans le cas de suppression de fichiers, de compression ou de sauvegarde de fichiers etc.), crontab sera la commande appropriée. Il est nécessaire d'avoir été autorisé par root pour utiliser cette commande.
crontab accepte deux options *-l* (pour lister les commandes déjà enregistrées) et *-e* (pour éditer la liste des commandes enregistrées. Attention édition sous vi par défaut, pour sortir :wq, modifiable par EDITOR).

```
$>crontab -l
20 3 8 * * /usr/local/bin/mon_script_de_sauvegarde
0  8 * * 0 /home/jpp/bin/recup_mail
```

Chaque ligne doit comporter 6 colonnes, si une colonne contient une astérisque, le champs devient non discriminant.

- 1ère : minutes (0−59)
- 2e : heures (0−23)
- 3e : jour du mois (1−31)
- 4e : mois (1−12)
- 5e : jour de la semaine (0−6,0 pour dimanche)
- 6e : nom de la commande ou du script

- Ou simplement passer un certain délai entre deux commandes, la commande *sleep X* permettra de différer de X secondes l'exécution de la commande suivante.

```
$>echo debut;sleep 10;echo fin
debut
fin
```

Pour surveiller toutes les anomalies sur votre station, la commande *top* (souvent installée par défaut) permet d'identifier les processus les plus consommateurs en temps CPU et en mémoire. En effet seuls ces deux facteurs peuvent réellement perturber votre station. Une consommation excessive

- de CPU empêchera le processeur de s'occuper de vous et de vos applications (tout sera figé ou extrêmement ralenti),
- de mémoire supérieure à la mémoire physique (12 Mo à 128 Mo) provoquera une pagination intense sur disque ralentissant les accès à la mémoire (puisqu'elle s'effectue alors sur disque avec un facteur 100).

DÉMO/TD (à faire avec plusieurs logins)

- Pour vous entraîner, utilisez la commande sous TP/TP2 commande_longue. Passez−la en arrière plan tout en la gardant active.
- Tuez−la avec un CTRL C.
- Lancez−la sans quélle se fasse tuer par votre déconnexion.
- Reconnectez vous et tuez−la par un kill.

11 L'élaboration de scripts

11.1 make

Lorsque l'on développe un programme (*prog.f*), il y a souvent plusieurs sous−programmes écrits dans de multiples fichiers (*fic1.f, fic2.f, fic3.f* etc). La phase de compilation consiste à générer les objets (les *.o*) et la phase d'édition de liens à créer un exécutable (souvent *a.out*).
Ces deux étapes ne doivent pas être faites systématiquement de manière complète. En effet si l'on modifie *fic2.f*, il ne faut recompiler que *fic2.f* (et pas les autres fichiers !) et penser aussi à refaire l'édition de liens.

Lorsqu'on a trois fichiers comme dans cet exemple, cela ne pose aucun problème, mais il est rare de faire tout un projet avec seulement trois fichiers. Il est plus fréquent d'avoir entre 30 et 50 fichiers sources.

De plus il peut exister de multiples dépendances dans les codes sources via les fichiers *include* (les .h ou les modules Fortran 95). La gestion de la compilation, si elle est faite à la main, peut générer des erreurs sournoises en introduisant des incohérences.

Le makefile résout tous ces problèmes (en se basant sur les dates de modification) et automatise le travail (certaines compilations peuvent durer plusieurs heures...). Voyons comment fonctionne un makefile sur notre exemple. En fait la commande *make* appelle par défaut un fichier *Makefile* (avec un M). Le travail consiste donc à "lister" dans le fichier *Makefile* les règles de dépendance, de compilation, les noms des fichiers et les options de compilation nécessaires.

Dans notre cas :

```
Ma_machine>cat Makefile
code : prog.o fic1.o fic2.o fic3.o
    f90 −o code prog.o fic*.o
prog.o : prog.f
    f90 −c prog.f
fic1.o : fic1.f
    f90 −c fic1.f
fic2.o : fic2.f
    f90 −c fic2.f
fic3.o : fic3.f
    f90 −c fic3.f
```

Remarques :

- les lignes contenant le caractère : sont appelée `règle`, à gauche de ce caractère se trouve la `cible`, à droite se trouvent les dépendances de la cible,
- en dessous de chaque de chaque règle se trouve les lignes de commandes qui va permettre au shell de passer des dépendance à la cible,
- les lignes indentées (lignes de shell) commencent par une tabulation (*TAB*).

Dans un projet complexe ayant plusieurs dizaines/centaines de fichiers, il est impensable d'écrire un fichier Makefile à la main, on utilise plutôt les règles implicites du make. Il est également possible d'utiliser des variables (ou macros). Ainsi au début du Makefile :
CF = f90 (si je souhaite changer de compilateur *CF=f77*)

FFLAGS=-g (si je ne souhaite plus déboguer *FFLAGS=*)
et même dans notre cas *OBJS=prog.o fic1.o fic2.o fic3.o*.
Ces macros doivent être utiliser via un *$()*

Notre exemple devient :

```
CF = f90
FFLAGS=-g
OBJS=prog.o fic1.o fic2.o fic3.o
code : $(OBJS)
   $(CF) -o code $(OBJS)
```

En fait

```
prog.o : prog.f
   $(CF) -c prog.f
fic1.o : fic1.f
   $(CF) -c $(FFLAGS) fic1.f
fic2.o : fic2.f
   $(CF) -c fic2.f
fic3.o : fic3.f
   $(CF) -c fic1.f
```

sont des règles implicites déjà connues de make !

Les dépendances : attention, si *fic2.f* contient *inc.h*, il faut que *fic2.o* dépende de *inc.h*, or par défaut make l'ignore... il faut donc rajouter :
fic2.o : fic2.f inc.h
Dans un vaste projet rajouter, à droite, à gauche des lignes dans le makefile n'est pas raisonnable.
Pour nous aider, nous allons nous appuyer sur les compilateurs qui disposent de l'option **-M**, permettant de générer les dépendances

```
Ma_machine$>cc -M fic2.f
fic2.o : fic2.f inc.h /usr/include/f_pvm.h
```

En fait *inc.h* dépendait lui même de *pvm.h* !
Attention c'est inefficace en Fortran 95 pour résoudre les dépendances de modules car la syntaxe `use` n'est pas interprétée par *cpp*)

Le makefile devient donc :

```
#Déclarations de variables
CF = f90
FFLAGS=-g
OBJS=prog.o fic1.o fic2.o fic3.o
MKDEP=-M
code : $(OBJS)
    $(CF) $(OBJS) -o $(@)
```

```
depend :
    $(CC) $(MKDEP) *.f > depend
include depend
```

Nous avons introduit une nouvelle règle *depend* qui ne dépend de rien, mais qui nous permet de faire *make depend* pour générer automatiquement le fichier *depend* qui sera rajouté (via include) au Makefile.

Remarques :

- le caractère @ permet de réutiliser le nom de la règle,
- le caractère \ permet de continuer la ligne de shell (attention pas de `blanc` après),
- le caractère # permet de d'inclure des commentaires,
- le caractère *:* dans une macro permet de faire des substitutions.

Il est possible de rajouter encore quelques règles utiles pour faire du ménage, créer une bibliothèque ou tout générer comme :

```
clean:      rm -f *.o ;\      echo j'ai détruit tous les *.o LIB=lib.a $(LIB) : $(OBJ)
ar -r $(LIB) $(OBJ) all: depend code      echo j'ai compilé tout ces fichiers : $(OBJ:.o=.f)
```

Attention : si vous souhaitez utiliser une variable d'environnement (comme $HOME, $TMPDIR) dans un makefile, il faut les faire précéder par $ (un autre $) comme suit :

```
install :
        cp $$HOME/source/*.f  $$WORKDIR
```

Pour déterminer quelles seront les actions effectuées par votre make (si vous avez un peu de mal à suivre les règles), sans les exécuter, utiliser l'option *-n* :
make -n all

Remarque : si vous avez changé un flag de compilation, et que la dernière compilation s'était bien passée, *make* ne fera rien ! En effet aucune règle lui impose de recommencer en cas de modification du fichier *Makefile*. Pour que *make* fonctionne, faites un *touch *.f** qui mettra la date des *.f* à la date courante.
Autre solution : l'effacement des .o et de l'exécutable obligera aussi le make à tout recompiler.

Les scripts

Il existe de multiples façons de faire des scripts, citons seulement awk, shell (ksh ou csh), perl etc.
Mais à quoi servent-ils ?
En fait ils sont vos meilleurs amis en cas de traitements inhabituels. Quelques exemples de traitements "inhabituels" :

1. substitution d'une variable dans un code, la variable TEMP doit être remplacée par Temp_Locale, TEMP apparaît à de multiples endroits et sur plusieurs dizaines de fichiers source, comment faire ?
2. liste des couleurs et de leurs occurrences dans un fichier html,

3. déplacer des fichiers suivant certains critères.

Mais avant de vous lancer dans l'écriture de scripts, soyez sûr que le traitement que vous souhaitez réaliser ne soit pas déjà effectué par une commande unix ! (ça arrive TRÈS souvent au débutant !). Aussi avant de voir plus en détail l'implémentation des exemples cités plus haut, passons en revue quelques commandes unix puissantes encore non citées dans ce cours.

sort : permet de trier les lignes d'un fichier (*fic*) suivant un champ particulier.

- *sort −r fic* : trie à l'envers
- *sort −n fic* : trie sur des valeurs numériques (123 est après 45)
- *sort −t: +n fic* : trie suivant le n+1 ^{e} champ (champs séparés par des :)

uniq : permet de supprimer les lignes identiques et de les compter (attention les lignes doivent être triées)

- *uniq −c fic* : compte les lignes identiques

paste : permet de "coller" deux fichiers l'un à côté de l'autre. Ne pas confondre avec l'un derrière l'autre (*cat fic >> fic2*)

- *paste fic1 fic2 > fic3*
- *paste −d\| fic1 fic2 > fic3* pour mettre un | entre les deux fichiers

cut : permet de "couper" une partie de ligne d'un fichier

- *cut −c 3−10 fic* : ne garde que du 3^{e} au 10^{e} caractère
- *cut −c −10 fic* : ne garde que du 1er au 10^{e} caractère
- *cut −c 3− fic* : ne garde que du 3^{e} au dernier caractère
- *cut −f 4,6 −d : fic* : ne garde que le 4^{e} et 6^{e} champs (: étant LE caractère de séparation, il doit être unique surtout avec le caractère "blanc")

tr : *tr b−z a−y*, sert à convertir le flux de caractères en remplaçant une lettre de la première liste par une autre de la deuxième liste (du même indice). Par exemple *tr a−z A−Z* permet de passer toutes les lettres d'un flux en majuscules.
Deux options intéressantes :

- *−s '\000'* qui supprime les caractères contigus passés en argument, ici les octets nuls,
- *−s ' '* qui supprime les caractères contigus blancs,
- *−d ':cntrl:'* qui supprime tous les caractères de contrôle,
- voir le man.

DÉMO/TD

- Recherchez la valeur identique (la plus grande possible et ayant la plus grande occurence) dans les fichiers commençant par `mpp` du répertoire TP/TP3.
- Et déterminez dans quels fichiers se trouve celle de plus grande occurrence.
- Réponse : echo "tpsu nqq* | vojr −d |tpsu −o | ubjm −o3 | dvu −d 7− | tpsu −o | ubjm −o1" | tr b−z a−y
- Réponse : echo "hsfq −o 281. nqq* | dvu −g 1 −e : | tpsu | vojr" | tr b−z a−y

Il est clair ici qu'un grand nombre de traitements peuvent être créés par la juxtaposition de commandes via des >, des | ou même des ; (l'enchaînement de commandes indépendantes peut être fait ainsi : *cd;ls*).

Si malgré toutes ces commandes, certaines choses restent impossibles à faire, nous devons construire un script. La première ligne du script doit (devrait) contenir *#!/chemin/shell*, afin de rendre votre script indépendant de votre shell courant, par exemple :
#!/usr/bin/ksh
ou
#!/usr/local/bin/perl
...
...
...

Les lignes qui précèdent sont des lignes de votre shell favori, un script sert seulement à mettre l'ensemble de ces lignes dans un fichier (*mon_premier_script*). Attention ce fichier doit avoir au moins les droits en exécution pour fonctionner. Lancez ensuite *./mon_premier_script*

11.2 ksh

Trois éléments indispensables du ksh vus aux travers d'exemples, les boucles, les tests, les variables.

Les boucles :

```
#!/usr/bin/ksh
for i in `ls *.f`
do
  echo je suis en train de traiter $i
done
```

Dans ce petit exemple, *ls *.f* fournit une liste d'indices à la boucle *for i*, (le ls est évalué car placé entre back quotes), le champ de la boucle étant limité par le couple *do/done*. Vous pouvez mettre n'importe quelle commande dès lors qu'elle génère une liste d'éléments du même ordre. *`cat fic1`* ou *`grep [0−9] fic1`* peuvent convenir ou simplement une liste comme *1 20 50*. Le traitement interne de la boucle peut être toute commande shell habituelle.

Pour les boucles sur des indices consécutifs on choisira une boucle *while* ainsi :

```
#!/usr/bin/ksh
i=1
while ((i<=5))
do
  qsub job.$i #Pour soumettre un job NQS sur une machine de calcul
  ((i=i+1))
done
```

Pour la lecture d'un fichier ou d'un flux (sortie de commande dans l'exemple) on pourra également choisir un *while* ainsi :

```
#!/usr/bin/ksh
cmd |
{
        while read ligne
        do
        echo $ligne
        done
}
```

Ou même en une seule ligne, on fait *commande2* tant que *commande1* est réussie

---

```
...
while commande1;do commande2;done
...
```

Les tests : sur les chaînes ou les fichiers [[ ]], ou sur les nombres (( ))

```
if (( numero < 10 )) || (( a!=2 ))
# liste d'opérateurs == != < > <= >=
then
echo "c'est plus petit"
else
echo "c'est plus grand"
fi
```

ou

```
read rep
if [[ $rep = "oui" ]]
#liste d'opérateurs = != < >
then
rm -f *.h
fi
```

ou

```
if [[ ! -a mon_fichier ]]
#liste d'opérateurs -a (any) -f (fichier) -d (directory) -S (non vide)
#fic1 -nt fic2 (vraie si fic1 plus récent)
then cp $HOME/mon_fichier .
fi
```

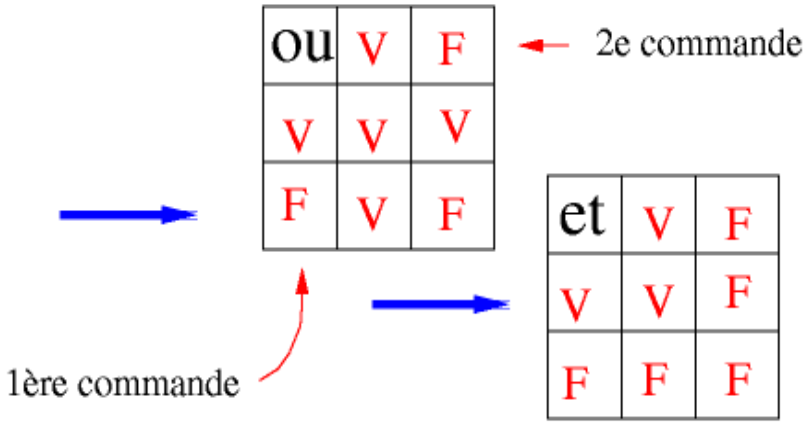
En cas de tests multiples, il est possible d'utiliser un bloc `case esac`

```
case $REP in
  [hH]elp  ) echo j'ai tapé help;;
```

```
  go|run) echo le code démarre;;
  *     ) echo le reste;;
esac
```

Il est possible de faire des tests plus simplement. En effet chaque commande exécutée renvoie un code de retour (0=vrai ou 1=faux) qui peut être interprété par le shell.

Pour évaluer une opération en binaire (0 ou 1, 0 et 0, etc.) le shell se contente du minimum. S'il peut déterminer le résultat de l'opération sans connaître la deuxième opérande, il ne va pas s'en priver !



Dans le cas d'un *ou*,

- si la première commande se passe "bien", il ne cherchera pas à faire la deuxième et donc ne l'exécutera pas.
- Si la première commande a généré une erreur, le shell est obligé d'exécuter la deuxième commande.

Dans le cas d'un *et* c'est le contraire !

Un *ou* est représenté par ||, et un *et* par *&&*. Voici des exemples appliqués au *ksh* :

- *grep XX file* **&&** *lpr file* qui permet d'imprimer un fichier s'il contient XX
- *ls mon_fichier* **&&** *echo "mon fichier est bien présent "*
- *mon_code* || *echo "le code a planté !"*
- *commande* || *exit* au sein d'un script

Les variables :

- *$?* permet de déterminer si une commande s'est bien "passée", son code de retour est alors *0*. Si elle a généré des erreurs, son code de retour est alors *>0*. Aussi n'est−il pas rare de voir dans un script :

```
...
commande_compliquee
if (($? != 0))
then
  echo "pb sur commande_compliquee "
  break
else
  echo "ça marche ..."
fi
...
```

Si vous faites un script avec des paramètres en entrée, comment les récupérer ?

- *echo $#* : pour connaître le nombre de paramètres
- *echo $@* : pour avoir la liste des paramètres

```
for i in $@
do
  echo $i
done
```

- *echo $0* : pour le nom de la commande
- *echo $n* : pour le paramètre de rang n ; il est possible de les décaler avec *shift* (sauf $0)
- *echo $$* : pour le numéro du processus courant (très utile pour créer des fichiers temporaires */tmp/poub_$$*)

Si vous souhaitez changer le contenu des $1, $2, $3 etc. utilisez *set −− val0 val1 val3 ...*

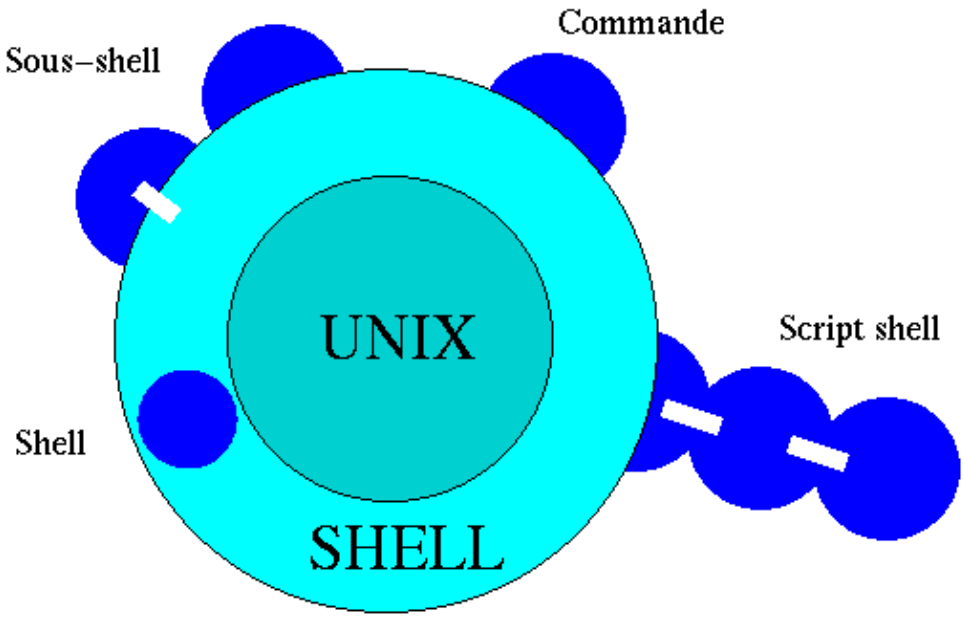
```
Ma_machine$>set -- 876 777 fin
Ma_machine$>echo $1 $2 $3
876 777 fin
```

Bien−sûr, rien ne vous empêche de créer vos propres variables (comme *i* ou *rep*). Comme vous l'avez vu dans les exemples précédents, une variable se définit par son *NOM* et s'appelle par *$NOM* ou *${NOM}* si vous voulez l'insérer dans une chaîne de caractères (*echo fic${i}ok*).

Attention à la visibilité de ces variables. Elles ne sont pas vues à l'extérieur de votre script sauf si :

- vous avez utilisé *export variable* :
  export rep
  dans ce cas, elles seront visibles dans un sous−shell (un script dans un script).

- ou vous dites explicitement à ce script de ne pas créer de sous−shell lors de son lancement via . (point)
  . mon_script



Vous comprenez pourquoi, dans les fichiers de configuration (comme *.profile*, *.kshrc* etc.) toutes les variables utiles à l'extérieur doivent être exportées et que ces scripts sont exécutés (ou réexécutés) via **.** (comme *. .profile)*.

Pour vous en persuader notez la différence entre set (positionnées) et env (positionnées et exportées).

Vous pouvez créer des sous−shells directement dans une ligne de commande en utilisant les parenthèses ( ) .

```
>pwd;(cd /;pwd);pwd
>/home/sos/jpp
>/
>/home/sos/jpp
```

Revenons aux variables afin de traiter quelques cas particuliers souvent rencontrés :

- dans le traitement des chaînes de caractères :
  l'utilisation de % (suppression du pattern à droite) et # (suppression du pattern à gauche) sur une chaîne *ma_chaine=/var/log/ma_log.old*
  - ♦ Le doublement de % ou # en ## et %% permet de supprimer la plus longue chaîne exprimée par le pattern.
  - ♦ pour ne garder que le nom sans le chemin

```
echo ${ma_chaine##*/}
ma_log.old
```

- pour changer le suffixe `.old` en `.new`

```
echo ${ma_chaine%.*}.new
/var/log/ma_log.new
```

- pour éliminer le chemin d'accès d'un nom de fichier

```
basename $ma_chaine
ma_log.old
```

- pour le calcul de variables :
  comme nous l'avons déjà vu, l'utilisation des ((et)) autour de l'expression numérique permet de l'évaluer. Les opérateurs sont ceux habituellement utilisés dans d'autres langages : / % − + * (% pour le reste de la division euclidienne).
  echo mon resultat est : $((354%54))

- pour l'affectation conditionnelle d'une variable : il est possible d'initialiser une variable uniquement si celle−ci ne l'est pas déjà ! C'est souvent utile pour des scripts système dans un environnement utilisateur, ou le contraire.

```
#lib positionnée par l'utilisateur
LIB_MATH=/usr/local/lib_test

#lib positionnée par le système dans un script
LIB_MATH=${LIB_MATH:=/usr/local/lib}

#le résultat est bien conforme au choix de l'utilisateur
echo $LIB_MATH
/usr/local/lib_test
```

- Attention *$(cmd)* n'est pas une variable mais le résultat d'une commande *cmd*.

```
Ma_machine$>ls
Ma_machine$>titi toto tata
Ma_machine$>result=$(ls tutu)
Ma_machine$>echo $result
Ma_machine$>The file tutu does not exist.
```

Vous pouvez également définir des tableaux à une dimension ainsi :

```
nom_du_tableau[indice1]=champ1
nom_du_tableau[indice2]=champ2
nom_du_tableau[indice3]=champ3
...
```

ou plus synthétiquement :

```
set +A nom_du_tableau champ1 champ2 champ3
```

Pour utiliser un tableau attention aux *{}* :

```
Ma_machine$>tab="zero"
Ma_machine$>tab[1]="premier"
Ma_machine$>tab[2]="deuxième"
Ma_machine$>echo $tab
zero
Ma_machine$>echo $tab[1]
zero[1]
Ma_machine$>echo ${tab[1]}
premier
```

La commande *eval* permet d'évaluer une ligne de commande

```
Ma_machine$>a='coucou'
Ma_machine$>b=a
Ma_machine$>echo $b
a
Ma_machine$>echo \$$b
$a
Ma_machine$>eval echo \$$b
coucou
Ma_machine$>c=\$$b
Ma_machine$>echo $c
$a
Ma_machine$>eval c=\$$b
Ma_machine$>echo $c
coucou
```

L'écriture d'un script comme celle d'un programme peut devenir assez complexe si l'on ne dispose pas d'outils pour suivre l'évolution du code/script.

En Ksh, en utilisant set −v ce dernier affichera toutes les commandes exécutées avant transformation et après transformation si l'on utilise set −x . Pour annuler ces effets remplacer le − par un +.

```
Ma_machine$>set -v
Ma_machine$>set -x
set -x
Ma_machine$>ls titi
ls titi
+ ls titi
titi
Ma_machine$>ls *
ls *
+ ls tata titi tutu
tata  titi  tutu
```

Comme dans de nombreux langages, vous pouvez créer vos propres fonctions. La syntaxe est la suivante :
nom_de_la_fonction () { liste; }
le blanc entre { et avant liste et le point virgule après liste sont obligatoires. Le code de retour est celui de la

dernière commande exécutée dans la fonction. Il n'y a pas de nouveau *process* créé. De plus toutes les variables de la fonction sont partagées sauf si l'attribut "local" est précisé.
local nom_de_la_variable_locale=456

```
Ma_machine$>cdl () { cd $1;ls -lrt; }
Ma_machine$>cdl Mail
Ma_machine$>inbox info mbox
Ma_machine$>pwd
Ma_machine$>/home/jpp/Mail
```

Remarque : ne jamais nommer son script "*test*", c'est une commande Unix !

Quelques variables purement shell peuvent aussi être utiles, *PRINTER* pour définir par défaut le nom de votre imprimante, et *SHELL* pour le nom de votre shell favori.

Pour finir, voici un exemple de script impossible à faire en une ligne ! Il consiste à sauvegarder toute une arborescence et la transmettre de manière sûre à une autre machine, c'est-à-dire en se prémunissant contre certains problèmes comme :

- l'impossibilité de lire le fichier (les droits ne sont pas bons ou le fichier n'existe pas !),
- un débordement d'espace (*file system full*),
- l'indisponibilité de la machine de sauvegarde,
- l'accessibilité de cette même machine,
- la corruption des sauvegardes,
- croire que tout va bien et ne pas être au courant d'éventuels problèmes.

```
#!/bin/bash
set -x
LOGIN_DEST=jpp
MACHINE_SAUV=mira
REP_A_SAUV=/home/jpp/poster
MON_FIC=/home/sos/jpp/sauv_$(date '+%m_%d_%y')
tar SPclf - $REP_A_SAUV 2>/tmp/$$res_tar | \
rsh -l $LOGIN_DEST $MACHINE_SAUV "dd of=$MON_FIC"
if [ $? = 0 ] && [ ! -a /tmp/$$res_tar ]
then
echo ça a marche !
rsh -l $LOGIN_DEST $MACHINE_SAUV "(mv sav1 sav2;mv $MON_FIC sav1)"
else
echo pb dans la sauvegarde | mail jpp@idris.fr
mail jpp@idris.fr < cat /tmp/$$res_tar
rsh -l $LOGIN_DEST $MACHINE_SAUV "(rm $MON_FIC)"
fi
rm /tmp/$$res_tar
```

Quelques lignes d'explications :

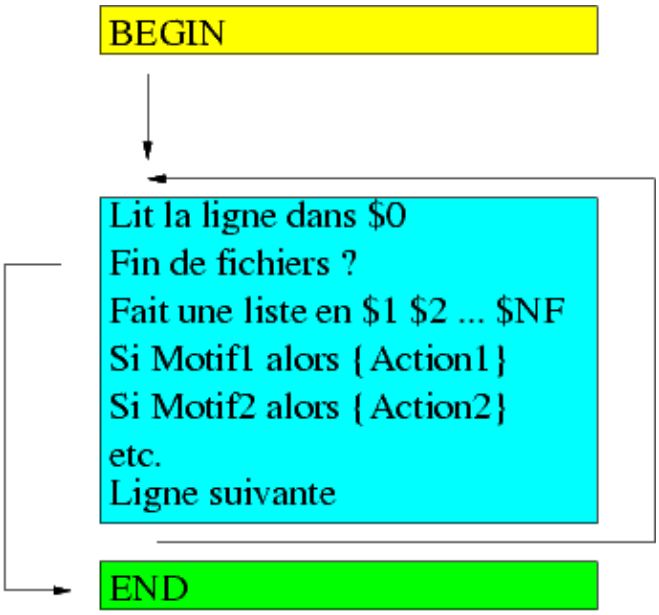
- date 'format' renvoie une chaîne de caractères représentant la date avec un certain format,
- l'option S du tar : permet de traiter les fichiers "troués",
- l'option P du tar : permet de ne pas supprimer le / en début de chemin,

- rsh : exécute une commande à distance sur `$MACHINE_SAUV` ,
- s'il y a eu un problème un mail est envoyé à l'auteur du script,
- le \ est le caractère de continuation de ligne.

11.3 awk

`awk` permet de traiter des fichiers ligne à ligne et d'y associer un même traitement. *awk* divise éventuellement le travail en trois étapes :

1. avant le traitement du fichier, via *BEGIN {print "avant"}*
2. pendant, via *{print "dedans"}* (chaque ligne du fichier traité exécute cette partie)
3. après, via *END {print x}*



Quelques exemples pratiques :

- Affiche le 2e et 4e champs d'un fichier fic1 (pour chaque ligne) avec un format
  awk '{printf ("%−9s | %3s \n",$2,$4)}' fic1

- Affiche les champs 3 et 2 si le champ 1 contient *URGENT* et compte les occurrences
  awk 'BEGIN{x=0} $1 ~ /URGENT/ {print $3, $2 ; ++x} END{print "total" x}' fic1

- Changement du séparateur de champs par défaut (" " ou TAB)
  awk 'BEGIN{FS=":"} print $1; print "la ligne complète est" $0' fic1

- Analyse des couleurs d'un document html (version 1)
  grep color unix_u_cours.html | awk 'BEGIN{FS="#"}{ print $2}' | cut −c1−6 | sort | uniq −c

- Analyse des couleurs d'un document html (version 2)
  awk 'BEGIN{FS="#"} $0 ~ /color/ {print substr($2,0,6) } ' unix_u_cours.html | sort | uniq −c

- Somme des éléments d'un fichier `poub`
  awk 'BEGIN{a=0} {a=a+$0} END{print a}' poub

- Moyenne des éléments communs d'un fichier `poub` (nom chiffre)
  awk 'BEGIN{NOM="";compteur=0;tot=0} { if ($1 == NOM) {tot=tot+$2;compteur=compteur+1} else { if (NR!=1) print tot/compteur,NOM ; NOM=$1;tot=$2;compteur=1} }END{print tot/compteur,NOM } ' poub

Chaque ligne du fichier traité est décomposée en champs ($1, $2, $3, ...), la ligne entière est référencée par $0. Deux autres variables internes sont très utiles, *NF* pour le nombre de champs et *NR* le numéro du "record" (lignes en général).

Une multitude de fonctions existent comme cos, int, exp, sqrt, toupper, tolower etc.

Cas des structures de commandes :

- les décisions (if, else) dans un bloc de commande
  awk '{if ($2=="") { print $3} else {print $2} } END {print "fini"}' fic1

- les décisions en dehors d'un bloc de commande
  awk 'NF>=7 { print $3 }' fic1
  awk '$2 ~ /toto/ || $1=="URGENT" { print $3 }' fic1

- les boucles (while, for, do−while)
  awk '{i=4;while (i
  awk '{for (i=11;i>=0;i−−) { print $i} }' fic1

Il est possible de créer des tableaux :

- mono−dimensionnés :
  Tab[87]="milieu"

- associatifs :
  Tab["rouge"]=924

Si votre script fait quelques lignes, mettez−le dans un fichier et utilisez l'option −f de awk ainsi :
awk −f mon_script fic1

Il est possible de faire des choses très complexes avec awk, mais si vous avez plus de trois lignes à faire en awk, je vous conseille de passer à perl.

11.4 perl

`perl` est un langage en soi, impossible de le décrire totalement ici. Il n'est pas en standard sur Unix (comme awk), il doit être installé. Cependant si vous avez la chance de l'avoir sur votre station, il peut, dans certains cas, vous permettre de faire des traitements assez complexes de manière simple. Voilà quelques "recettes de cuisine" utiles :

- *perl −p −i −e "s|\bOLDNAME\b|NEWNAME|g;" `ls *.f *.F`*
  permet en une seule ligne de changer OLDNAME en NEWNAME sur tous les fichiers sources

Fortran du répertoire, et de répondre au cas 1 cité plus haut.

- pour ne récupérer que le texte situé entre deux mots (`start` et `end`) même s'ils ne sont pas sur la même ligne !
  perl −e "while (<>) {print if /start/ .. /end/}" fic

- la même chose mais pour les commentaires en C
  perl −e "while (<>) {print if /\// .. /*\//}" fic*

Perl peut utiliser des *packages* d'interface (modules) permettant d'accéder aux réseaux, aux bases de données, aux analyseurs syntaxiques, etc. et faire ainsi en 3 lignes ce qui nécessiterait autrement plusieurs milliers de lignes.

Voici un exemple de script utilisant un module réseau :

```
#!/usr/local/bin/perl −w

use Net::FTP;
$ftp = Net::FTP−>new("truc.labo.fr");
$ftp−> login("login1","qpap?1");

$ftp2 = Net::FTP−>new("machin.labo.fr");
$ftp2−> login("login2","qpap?2");

$pwd = $ftp−>cwd("conftempo");
$pwd2 = $ftp2−>cwd("respconftempo");
$ftp−>type("I");
$ftp2−>type("I");

@tab = $ftp−>ls();
foreach $a (@tab) {
     $ftp−>get($a);
     print ($a," recup ok\n");
     $ftp2−>put($a);
     print ($a," transfert ok\n");
     $tmp=`rm −f $a`; }
$ftp−>quit;
$ftp2−>quit;
print ("fin du transfert");
```

11.5 cpp

C'est le préprocesseur du C (donc disponible partout). Il exécute des instructions particulières appelées directives. Ces directives sont identifiées par le caractère # en tête. Ces directives sont de trois sortes :

1. Les pseudo−constantes ou pseudo−fonctions: #define identificateur [chaîne−de−substitution]

```
#define  TAILLE 256
#define  TAILLE_EN_OCTETS TAILLE*sizeof(int)
#define NB_ELEMENTS(t) sizeof t / sizeof t[0]
```

```
main()
{
   int  tab[TAILLE];
   int  i;
   for(i=0; i < TAILLE; i++) tab[i] = i;
   printf("Le tableau tab contient %d octets\n",TAILLE_EN_OCTETS);
   printf("Le tableau tab contient %d octets\n",NB_ELEMENTS(tab);
}
```

Pseudo−constantes déjà définies *__DATE__* pour la date du jour, et *__TIME__* pour l'heure.

2. La directive *#include fichier_a_inclure* permet d'insérer le contenu d'un fichier dans un autre.
3. Les directives *#ifdef* et *#ifndef* permettent de tester l'existence d'une pseudo−constante.

```
#ifdef identificateur
  partie-alors
[#else
  partie-sinon]
#endif
```

Exemple *cpp prog.c > prog_a_compiler.c* :

```
#ifdef DEBUG
  printf("je suis dans la boucle a l'indice %d \n",i);
#endif
```

Pour avoir un préprocesseur encore plus souple utilisez gpp (generic preprocessor http://auroux.free.fr/prog/gpp.html), vous pourez ainsi définir vos propres macros et syntaxes.

DÉMO/TD

- Comment mettre en commentaire 100 lignes de code C sans utiliser les */ et /* à chaque ligne ?
- Sous TP/TP4.1 créer des images gif animées (comme celles utilisées pour ce cours) par un script.
- Générer des courbes xmgr d'après l'analyse/le traitement de fichiers de log sous TP/TP4.2

12 Travailler sous X ...

12.1 X

Le système de multifenêtrage X11, aussi appelé X window System, a été développé au MIT et est aujourd'hui l'interface graphique de la quasi totalité des ordinateurs sous Unix.

Il permet de rendre totalement indépendants plusieurs aspects d'un système de fenêtrage et ainsi de développer des applications clientes des services de l'interface, indépendamment du matériel, du système d'exploitation, de la localisation de l'interface et de l'aspect de celle−ci.

Ce système rend donc également possible l'affichage de clients en provenance de plusieurs machines sur un écran unique (le serveur). Par dessus X11 viennent ensuite se greffer des window−managers ou gestionnaires d'interface qui determinent l'aspect et le mode d'interaction.

La terminologie X11 peut sembler contradictoire. En effet, un gestionnaire d'écran, comme une application, sont considérés comme des clients, et l'écran physique est lui considéré comme un serveur. En fait, il est assez normal que la couche offrant des services graphiques utilisés par des clients distants se nomme serveur. Le serveur est donc la machine sur laquelle l'affichage a lieu. Tout le reste n'est que client, y compris le gestionnaire d'interface. Les deux parties (client et serveur) peuvent très bien se trouver sur une même machine.

J'espère que vous travaillez déjà sous X ! Mais il est bon de rappeler que si c'est assez simple d'ouvrir une fenêtre X en local sur votre écran, il est un peu plus complexe d'afficher chez vous la fenêtre d'un code tournant sur une autre machine. Le système X11 étant distribué, les clients peuvent donc venir de n'importe quelle machine du réseau, et sur ces machines de n'importe quel utilisateur. La couche de sécuritéde X11 a pour but de n'autoriser l'accès qu'a des machines ou des utilisateurs spécifiques.

Prenons le cas de votre laboratoire et de l'IDRIS. Vous travaillez chez vous et avez ouvert une fenêtre (suivie d'un rlogin ou d'un telnet) vers l'IDRIS. Dans cette fenêtre tourne un débogueur, un éditeur ou autre. Si ces outils sont X, vous obtiendrez inévitablement un message d'erreur comme suit :
Error: Can't open display:

Vous devez alors faire deux choses :

1. *export DISPLAY= ma_machine.mon_labo.fr:0* sur la machine de l'IDRIS, afin qu'elle sache où renvoyer cette fenêtre (`DISPLAY` est une variable du shell). A ce stade si vous essayez de nouveau vous obtiendrez un nouveau message :
   Xlib: connection to "ma_machine.mon_labo.fr:0.0" refused by server
   Xlib: Client is not authorized to connect to Server
   Error: Can't open display: ma_machine.mon_labo.fr:0

2. *xhost + machine_IDRIS* , sur votre machine locale, dans la fenêtre console, afin d'autoriser *machine_IDRIS* à s'afficher chez vous.

(Attention, ceci n'est pas complètement sans danger, utilisez plutôt xhaut pour plus de sécurité).

Dès lors vous devriez pouvoir afficher des fenêtres X chez vous.
Attention, restent encore deux pièges dans certaines universités :

1. il se peut que la connexion soit extrêmement lente (problème de bande passante), une fenêtre X peut mettre 5 minutes à arriver ! Pas de solution à part demander une augmentation du débit de la ligne.

   Ceci dit pour en être bien sûr, *traceroute* peut vous dire s'il y a un problème entre votre machine et la machine destination.

   - cas d'une ligne saturée :

```
Ma_machine>traceroute www.koi.com
traceroute to www.koi.com (205.199.140.38)
 1  ipsgw2 (130.84.12.1)  2 ms  1 ms  1 ms
 2  ipsctl (130.84.164.1)  2 ms  1 ms  1 ms
 3  ipsgw1 (130.84.160.2)  2 ms  2 ms  2 ms
 4  ea-rerif-atm (192.54.202.19)  2 ms  2 ms  2 ms
 5  stlambert1.rerif.ft.net (193.48.53.213)  239 ms  13 ms  110 ms
 6  stamand1.renater.ft.net (193.48.53.9)  13 ms  4 ms  3 ms
 7  rbs2.renater.ft.net (195.220.180.18)  4 ms  3 ms  3 ms
 8  paii.renater.ft.net (195.220.180.29)  5 ms  7 ms  8 ms
 9  pos2.opentransit.net (193.55.152.25)  6 ms  5 ms  6 ms
10  pos6.opentransit.net (193.55.152.90)  138 ms  138 ms  138 ms
11  pos1.opentransit.net (194.206.207.54)  138 ms  138 ms  138 ms
12  sl-stk-1-2.sprintlink.net (144.232.4.29)  138 ms  137 ms  140 ms
13  sea-7-0.sprintlink.net (144.232.9.86)  149 ms  146 ms  146 ms
14  sea-8-0-0.sprintlink.net (144.232.6.54)  319 ms  147 ms  147 ms
15  semacorp-2-0-0.sprintlink.net (144.228.96.18)  158 ms  199 ms  189 ms
16  10btin.aa.net (205.199.143.17)  195 ms  177 ms  195 ms
17  www.koi.com (205.199.140.38)  208 ms  204 ms  244 ms
```

   - cas d'une ligne quasiment bloquée :

```
Ma_machine>traceroute ftp.lip6.fr
traceroute to nephtys.lip6.fr (195.83.118.1), 30 hops max, 40 byte packets
 1  ipsgw2 (130.84.12.1)  2 ms  2 ms  1 ms
 2  ipsctl (130.84.164.1)  2 ms  2 ms  1 ms
 3  ipsgw1 (130.84.160.2)  2 ms  2 ms  2 ms
 4  ea-rerif-atm (192.54.202.19)  2 ms  2 ms  2 ms
 5  stlambert1.rerif.ft.net (193.48.53.213)  3 ms  3 ms  3 ms
 6  u-jussieu-paris-atm.rerif.ft.net (193.48.53.198)  6 ms  4 ms  6 ms
 7  r-jusren.reseau.jussieu.fr (134.157.255.126)  80 ms  543 ms  731 ms
 8  * * *
 9  * * *
```

2. il se peut aussi que la connexion ne se fasse tout simplement pas ; dans ce cas c'est souvent un problème de filtre en interne chez vous (voyez votre ingénieur système / réseaux).

   Pour vérifier un simple *ping machine_b* vous permettra de savoir si *machine_b* est accessible ou pas !

   - cas accessible (demande de trois échos):

```
Ma_machine>ping -c 3 www.ibm.com
PING www.ibm.com: (204.146.18.33): 56 data bytes
64 bytes from 204.146.18.33: icmp_seq=0 ttl=235 time=131 ms
64 bytes from 204.146.18.33: icmp_seq=1 ttl=235 time=125 ms
64 bytes from 204.146.18.33: icmp_seq=2 ttl=235 time=133 ms
```

- cas d'indisponibilité (www.ibm.com n'est pas dans les filtres.)

```
Ma_machine:>ping www.ibm.com
PING www.ibm.com: 56 data bytes


blocage --> sortie avec un ^D
```

13 Un peu de sécurité

Si votre compte est piraté, vous risquez de :

- perdre du temps pour analyser les dégâts,
- perdre vos données,
- perdre du temps pour récupérer vos données,
- perdre le système de votre machine,
- perdre du temps à tout réinstaller,
- perdre la confiance de votre entourage, et vous risquez une décrédibilisation de vous même et de votre organisation,
- rendre des comptes lors d'affaires judiciaires, lorsque votre responsabilité est reconnue.

13.1 passwd

Le changement des *passwd* peut être laborieux, surtout si vous avez un compte sur plusieurs machines ; il l'est d'autant plus qu'il est assez fréquent (changement toutes les 10 semaines ; peut varier suivant les sites). Si la contrainte du renouvellement du passwd ne peut être évitée pour des raisons de sécurité, la répétition sur n machines peut être supprimée grâce à *Passwd* (avec un P). La commande *Passwd* est disponible sur Rhodes (et donc uniquement à l'IDRIS) et propagera votre login (qui doit être inférieur à 8 caractères) sur l'ensemble des machines de l'IDRIS. Sachez que des commandes standard existent comme `yppasswd` pour faciliter la propagation des mots de passe.

Comment faire pour trouver un bon mot de passe ?
Surtout ne pas utiliser son nom, son prénom et même aucun prénom ou nom classique. Les pirates utilisent des dictionnaires pour "cracker" les mots de passe, alors bannissez tous les mots du dictionnaire même étrangers ! De plus maintenant avec un "simple" ordinateur portable un mot de passe de moins de 8 caractères est "cassable" en moins de deux jours.

Alors comment faire ?
Il suffit de créer des mots à vous ! le plus simple est de créer une phrase que vous affectionnez particulièrement ou qui vous touche en ce moment et d'en extraire les premières lettres de chaque mot, c'est très efficace et inviolable. Évitez aussi les caractères spéciaux, certaines machines n'apprécient pas !

Par exemple :
un cours unix en préparation en 98 devient : *1cuepe98*
Il fait toujours mauvais a Paris ! : *IftmaP!*

Remarque : ne pas utiliser des caratères comme # ou @ dans votre mot de passe car ils sont parfois interprétés par certaines commande comme telnet.

13.2 s[cp|login|sh]

Les `s`commandes offrent les mêmes fonctionnalités que les `r`commandes mais en beaucoup plus sûres, en imposant un cryptage et une authentification plus robuste.

Comment fonctionnent−elles ?

- L'installation : il faut d'abord générer le couple clés `publique` / `privée` par *ssh−keygen* sur votre machine L (comme Locale). Cette commande vous demandera de composer "`une phrase`", mettez ce que vous voulez mais souvenez−vous en !

  La clé `publique` (contenue dans *.ssh/identity.pub*) peut être donnée à tout le monde, mais la clé `privée` devra rester stockée sur L. Vous pouvez donc envoyer ou recopier la clé `publique` où vous voulez, (il faut la recopier sur la machine distante D (comme Distante) dans le fichier *.ssh/autorized_keys*).

- L'utilisation : connectez−vous sur D à partir de L (en rentrant "`la phrase`") via les commandes
  - *ssh −l mon_login D.labo.fr ls*
  - *slogin −l mon_login D.labo.fr*
  - *scp fic mon_login@D.labo.fr:*

  Ces commandes entrent alors en contact avec le serveur ssh de D, qui va envoyer un *challenge* encrypté avec la clé `publique` qu'il aura trouvé sur D dans le fichier *.ssh/autorized_keys*. Si L arrive a décrypter le *challenge* (uniquement possible si L possède la clé `privée` −− la clé `publique` ne peut pas décrypter, mais seulement crypter), L renvoie le *challenge* décrypté à D prouvant ainsi qu'il a bien la clé `privée` ce qui l'authentifie.

- L'automatisation : pour éviter de taper la phrase à chaque fois (très gênant dans les scripts), tapez ssh−agent ksh; ssh−add
  Vous pourrez alors utiliser ssh, *scp* ou *slogin* directement sans aucun contrôle. Attention, si la machine L est piratée toutes les machines accessibles par une `scommande` sont alors aussi compromises.

14 Divers

14.1 Voir des choses cachées

Il est possible de découvrir :

- les chaînes de caractères au sein d'un programme/binaire non visible (non éditable) par la commande *strings*,
  strings a.out

- les caractères spéciaux (NL, CR, BS etc.). Si vous avez des soucis (comme un comportement curieux) avec des fichiers ou des noms de fichiers, il est possible qu'un caractère spécial se soit inséré dans une chaîne de caractères.

  Pour éliminer ce doute, utilisez *od* qui vous permettra de visualiser TOUS les caractères d'un fichier sous forme ASCII :
  od −c fic_avec_problème
  Les options `-vte` de cat permettent également de visualiser les caractères non imprimables.

```
Ma_machine>ls f*  | more
fin1.gif
fmdictionary
Ma_machine>ls f* |od −c
0000000   f   i   n   1   .   g   i   f  \n   f   m   d   i   c   t   i
0000020   o   n   a   r   y  \n
0000026
Ma_machine>ls f* | cat −v
fin1.gif
fmdictionary
```

- Pour les archives : vous avez récupéré ou avez créé une archive en Fortran ou C et votre programme n'arrive pas à trouver votre sous−programme défini normalement dans cette archive. Comment voir le nom des appels (des sous−programmes) ?
  ar −tv ma_lib.a

- Pour les objets : vous disposez d'un ***.o***, mais ignorez son contenu, faites un :
  nm mon_objet.o

14.2 Ou le contraire, ne pas les voir

Supprimer des choses indésirables comme les balises de style des *man* qui tabulent ou passent en gras certains caractères. Essayez d'imprimer un man, vous allez tout de suite comprendre...

```
Ma_machine:>man ls | cat −v
```

```
LS(1)                Cray Research, Inc.     SR-2011 9.0
N^HNA^HAM^HME^HE
     l^Hls^Hs - Lists contents of directory
S^HSY^HYN^HNO^HOP^HPS^HSI^HIS^HS
     l^Hls^Hs [-^H-1^H1] [-^H-b^Hb]
     [-^H-B^HB] [-^H-c^Hc] [-^H-e^He]
     [-^H-f^Hf] [-^H-F^HF]
...

Ma_machine:>man ls
LS(1)      Cray Research, Inc.     SR-2011 9.0
NAME
     ls - Lists contents of directory
SYNOPSIS
     ls [-1] [-a] [-A] [-b] [-B] [-c] [-C] [-d] [-e] [-f]
     [-k] [-I] [-l] [-L] [-m] [-n] [-o] [-p] [-P] [-q] [-r]
     [-u] [-x] [file ...]
...
```

Afin d'éviter d'avoir la première sortie (avec tous les ^H et ^C) sur votre papier, utiliser *col -b* ainsi :
man ls | col -b | lpr
La commande *col* agira comme un filtre prenant à gauche (*standard in*) un flux "pollué" venant de *man* et délivrant à droite (*standard output*) un flux débarrassé des différents caractères de contrôle à la commande d'impression *lpr*.
Une autre option utile de la commande est *-x* qui transforme les TAB dans un fichier en "blanc".

14.3 Commandes inclassables ou plutôt système

- pour créer instantanément un fichier de grande taille (mais vide !, c'est utile pour des tests)
  >*dd if=/dev/zero of=empty_file bs=1024k count=10*

- pour connaître l'occupation des files systems
  >*df -k*

- pour connaître l'occupation d'un répertoire et de ses sous-répertoires
  >*du -k .*

- pour monter ces systèmes de fichiers (avec /etc/fstab à jour)
  >*mount -a*
  Si c'est un nouveau système de fichiers (une disquette par exemple)
  >*mount /dev/fd0 /mnt/floppy*

- pour sectionner un fichier en plusieurs parties, utile pour mettre un fichier supérieur à 1.4M sur une disquette. `split` générera autant de fichiers que nécessaire, la commande crée des fichiers de la forme xaa, xab, xac ... xba, xbb etc.

```
>split -b 1400k gros_fichier
>ls x*
xaa xab xac xad xae xaf
>
```

Pour reconstituer le fichier original, *cat xa* > mon_gros_fichier*

- pour coder un fichier binaire afin de la transmettre par mail (non mime)
  >*uuencode< fichier_a_coder>fichier_codé* #pour le codage
  >*uudecode <fichier_codé* #pour le décodage et sauvegarde dans le fichier originel

14.4 Quelques références bibliographiques

- `sed & awk` ISBN: 0−937175−59−5 chez O'Reilly & Associates, Inc.
- `Unix in a nutshell` ISBN: 1−56592−001−5 chez O'Reilly & Associates, Inc.
- `KornShell Programming Tutorial` ISBN: 0−201−56324−x chez Addison−Wesley
- `Perl` ISBN: 0−937175−64−1 chez O'Reilly & Associates, Inc.
- `Teach Yourself PERL in 21 Days` ISBN: 0−672−30586−0 chez SAMS PUBLISHING

Bon travail en mettant en pratique tous ces conseils et astuces.