GUIDE

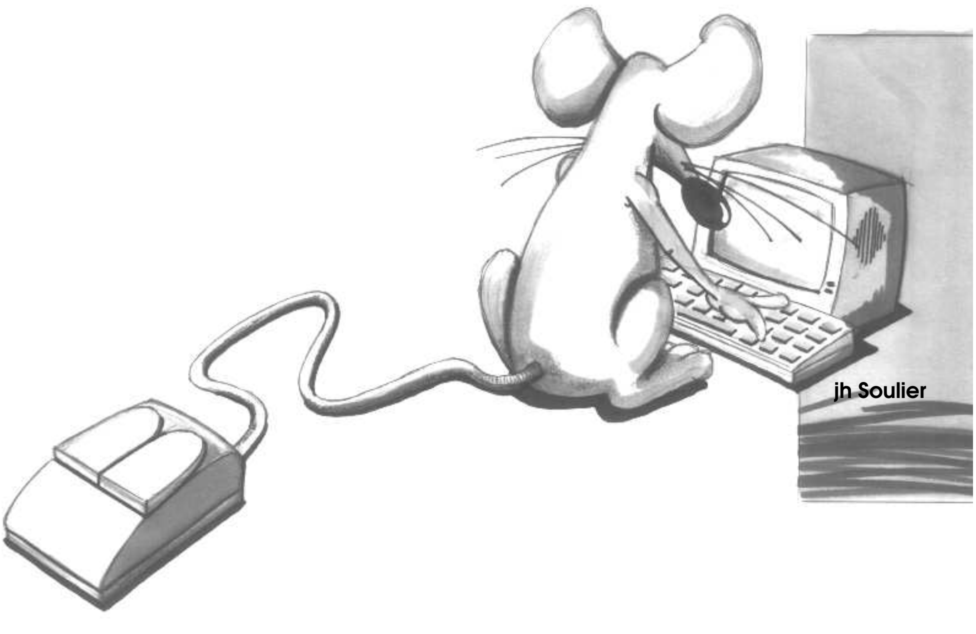
du

KornShell

sous

UNIX

(HP-UX 8.0, SUNOS 5.1, AIX 3.2)



Jean François Pujol

août 93

INTRODUCTION

Ce document constitue une aide à la programmation du KornShell (ksh), et non un manuel de référence; c'est pourquoi il est destiné aux utilisateurs ayant déjà des connaissances de base d'un shell (sh ou csh).

Il contient des exemples qui se veulent instructifs, et pas une base rigide de programmation : vous y piocherez donc ce qui vous semble bon.

Tous les exemples ont été exécutés à l'origine sur un hp9000 série 400, sous HP-UX 8.0, avec une version du KornShell datée du 16/11/88.

La version HP-UX 9.0 n'a apporté que de légères modifications au niveau des commandes (convergence vers POSIX 1003.2), mais n'a pas été testée ici spécifiquement. Dans le cas où une incompatibilité apparaîtrait, le mieux est de se référer aux spécificités SUN ou IBM: il est probable que l'une d'entre elles permette de comprendre le phénomène. Il faut savoir que HP fournit le shell POSIX (appelé `sh`, comme le BourneShell) à partir de la version 9.0, et que le shell POSIX est presque une copie conforme du KornShell.

Pour tenir compte des utilisateurs travaillant sous SUN-OS 5.X (SOLARIS 2.X), ou AIX 3.2, les exemples ont été testés également dans ces environnements, et les différences qui apparaissent par rapport à la version HP-UX figurent en annexe.

Pour SUN-OS 5.1, la plate-forme était une "SparcStation 1+", et pour AIX 3.2, une station RS6000 modèle 340, avec un clavier (et des messages) en version française.

voir

Dans la suite du document, un symbole dans la colonne signifiera que l'exemple en cours n'est pas directement adaptable dans l'environnement SUN-OS: il faudra se reporter au chapitre adéquat pour y retrouver les particularités;

voir

De la même façon, un symbole correspondra à une remarque spécifique à AIX.

Le fait que la version de base soit écrite pour HP-UX n'est en rien lié à une quelconque norme ou référence: le choix est arbitraire.

Surtout, n'hésitez pas à me communiquer vos remarques : cela me permettra de corriger les imperfections, qui, je n'en doute pas, doivent encore émailler ce document.

Guide d'utilisation du guide

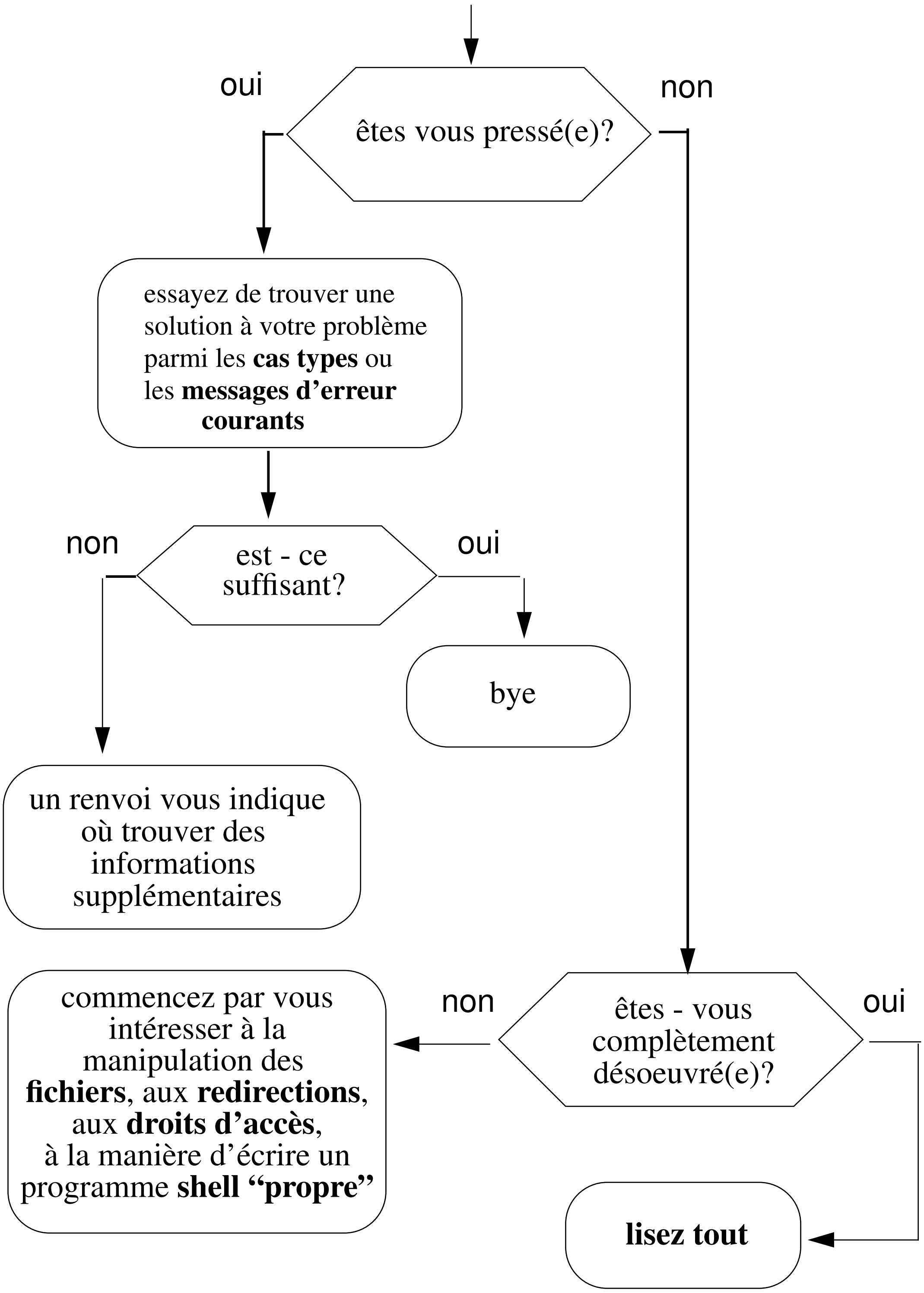


Table des Matières

I.

I. Description

I.1. Introduction : qu’est ce que le shell?

I.1.1. Description

Le shell est un interpréteur de commandes. Il permet de lancer les commandes UNIX disponibles sur votre station, en leur affectant un certain nombre de paramètres d’exécution, de contrôler les données d’entrée et de sortie, les messages d’erreur éventuels, et surtout de les enchaîner de manière efficace et pratique.

Un shell est démarré pour chaque console ouverte (ou fenêtre X11 équivalente créée) ; il est intéractif dans ce cas, car les lignes de commande tapées au clavier sont exécutées une par une avec confirmation systématique (touche <return>).

Un shell est également créé chaque fois que la commande à exécuter se trouve être un fichier ascii : le shell lit successivement toutes les lignes du fichier, supposant que ce sont des commandes ; il disparaît (“rend la main”) une fois la fin de fichier atteinte.

I.1.2. Algorithme simplifié

Si le shell est intéractif, on peut représenter son fonctionnement sous la forme de l’algorithme simplifié suivant :

```
TANT QUE < durée de la connexion>
      lire_ligne_au_clavier(ligne)
      traiter_ligne(ligne)
      afficher_resultats()
      afficher_prompt()
FIN TANT QUE
```

Si le shell n’est pas intéractif, alors on peut considérer qu’il est lancé directement à travers la fonction traiter_ligne() dont le paramètre est le nom du fichier à traiter ;

I.1.3. Voici l’algorithme de la fonction traiter_ligne() :

```
traiter_ligne(ligne)
    POUR <premier champ de la ligne> DANS
        * <nom de fichier ascii> :
            TANT QUE <fichier ascii> NON VIDE
            traiter_ligne(<ligne courante du fichier ascii>)
            FIN TANT QUE
        * <nom de binaire executable> :
            charger_et_lancer(<fichier binaire>)
        * <commande shell> :
            executer_commande()
        * AUTRE :
            afficher_message(“command not found”)
    FIN POUR
```

I.1.3. Les différents shells disponibles

Le shell est lui même un fichier binaire, qu’il est possible de lancer comme n’importe quelle autre commande UNIX. Ainsi, lorsqu’on parle du “shell”, cela recouvre en fait un ensemble de trois interpréteurs différents, possédant chacun un exécutable spécifique :

i. Le BourneShell, (commande “sh”), est l’original, le plus ancien ayant existé sous UNIX ; c’est celui qui possède aussi le moins de fonctionnalités, et qui est le moins pratique à utiliser ; par contre, on est sûr de le trouver quelque soit le type de machine.

ii. Le Csh (commande “csh”), est plus récent et intègre des fonctions supplémentaires comme l’historique des commandes, indispensable en usage courant ; on le trouve en standard sur la plupart des machines ; par contre sa syntaxe n’est pas compatible avec le Bourne Shell ;

iii. Le Ksh (commande “ksh”), est le dernier né et le plus complet ; il accepte toutes les commandes du Bourne Shell, possède un historique de commandes perfectionné, et la plupart des fonctionnalités du Csh.

C’est ce dernier qui servira de base pour les exemples qui suivent, bien que beaucoup de remarques soient valables quelque soit le shell utilisé.

I.2. Qu’est ce qu’un processus UNIX ?

I.1.3.

Un processus UNIX est un fichier binaire en train de s’exécuter avec un certain environnement :

- des pointeurs sur les fichiers ouverts ;
- des réactions programmées aux signaux :
    - division par zéro
    - perte de connexion avec la console
    - réception du signal INTERRUPT (touche CTRL C)
    - ...
- etc...

Pour démarrer un nouveau processus, c’est à dire chaque fois que l’on veut lancer une nouvelle commande (grep, sort, compile, ..), une technique standard est appliquée ;

- Le processus en cours fait appel à une fonction spéciale qui lui permet de se recopier lui-même intégralement. Tout est recopié exactement en double, la zone mémoire, la zone de code, les pointeurs sur les fichiers, etc... Seul, le code de retour de la fonction permet de différencier l’original (le père) de la copie (le fils) ;
- l’environnement du processus fils est modifié en partie si nécessaire (descripteurs de fichier, variables exportées, signaux,...), puis la zone de code du fils est écrasée avec le code du nouveau programme à exécuter, qui démarre ;
- le père attend la fin du fils, puis continue de s’exécuter.

Cette méthode peut paraître complexe, mais c’est exactement ce qui se passe chaque fois que le shell lance une commande (par la fonction charger_et_lancer() de notre pseudo-interpréteur vu plus haut).

On y découvre à l’occasion la raison pour laquelle le vocabulaire lié aux processus est souvent tiré de tout ce qui touche la descendance ; c’est une constante : tout processus a été lancé par un processus père (sauf le premier !) et il hérite de la plupart des ses caractéristiques.

Si votre shell possède un paramétrage dont vous ne savez attribuer la provenance en étudiant vos fichiers de configuration, il y a de grandes chances que ce soit un processus “ancêtre” qui l’ai positionné.

Mais la création d’un nouveau processus est une action coûteuse en ressources système, et nous verrons plus loin quelles sont les règles à appliquer pour éviter d’y faire trop souvent appel lors de l’exécution de programmes.

I.3.1.

I.3. Format d’une commande UNIX élémentaire

I.3.1. Description

Si l’on oublie pour l’instant les caractères de contrôle du shell comme ‘ ;’ ‘|’ ou ‘&’, et la syntaxe de certaines commandes internes au shell (affectation, tests, boucles for,...), une ligne de commande élémentaire a la forme suivante :

<champ0> <champ1> <champ2> ... <champN>

les champs étant des suites de caractères terminés par des <espace>, ou des <tabulation>. Alors :

- le <champ> 0 est supposé être le nom d’un fichier binaire ou ascii, ou le nom d’une commande interne au shell ;
- les <champ> 0 à N sont des paramètres qui seront passés au démarrage à l’exécutable <champ0> (comme un appel de fonction).

Le shell va essayer de transformer chaque ligne de commande de manière à retomber sur ce schéma :

- le premier champ de la ligne prend le rôle de l’exécutable à lancer ;
- tous les arguments (de 0 à N) sont “empilés” lors de l’appel.

Au programme appelé de s’occuper correctement de ses paramètres, en faisant le tri, et en les utilisant de manière souhaitée.

I.3.2. Exemples

exemple 1 : `grep -i -v TOTO /tmp/essai`

Cette commande est destinée à rechercher dans le fichier /tmp/essai toutes les lignes qui ne comprennent pas (option -v) la chaîne de caractères “toto” apparaissant indifféremment en majuscules ou minuscules (option -i).

Cette commande se décompose en :

<champ0>	= “grep”	: la commande
<champ1>	= “-i”	: le premier argument
<champ2>	= “-v”	: le second
<champ3>	= “TOTO”	: le troisième
<champ4>	= “/tmp/essai”	: le quatrième

Au moment de l’appel de `grep`, aucune différence n’est faite entre les quatre arguments ; le shell lance l’exécutable `grep` en lui fournissant les quatre arguments, sans se préoccuper de leur ordre, ni de leur valeur ou signification.

Par contre, lorsque la commande grep commence à “travailler”, sa première action est de faire le tri et de tester les paramètres qui lui été fournis. C’est à ce moment que le type des arguments a une signification : certains doivent commencer par un tiret (ce sont des “options” de la commande) et d’autres non (ce sont des “paramètres” de la commande). I.3.2.

En résumé, le shell sépare et fournit à la commande les différents champs constituant la ligne ; la commande récupère les champs et vérifie qu’ils correspondent bien à sa syntaxe.

<u>**exemple 2**</u> : `sed 's/^#//' /tmp/toto`

Cette commande renvoie le fichier `/tmp/toto` sans caractère “#” en début de ligne.

Elle se décompose en :

<champ0>	= “sed”	: la commande
<champ1>	= “s/^#//”	: le premier argument
<champ2>	= “/tmp/toto”	: le second

On voit ainsi que la commande `sed` utilise le premier argument comme la commande à exécuter sur le fichier dont le nom est indiqué par le second argument.

I.3.2.

I.4. Les entrées-sorties d’une commande UNIX élémentaire

Pour faire dialoguer une commande (un processus) avec l’extérieur, et le shell en particulier, on dispose de différentes solutions ;

en ENTREE : **Les paramètres de position[1]:**

de 0 à N, comme on a vu au paragraphe précédent. (le nom de la commande est passé aussi en tant qu’argument 0).

Si la ligne de commande est :

```
cmd arg1 arg2
```

l’appel de `cmd` pourrait être assimilé à l’appel d’une fonction cmd() avec les 3 arguments “cmd”, “arg1”, “arg2” passés par valeur ;

Les variables exportées :

Toutes les variables faisant partie de l’environnement exporté sont visibles et utilisables par la commande ; par contre, la modification du contenu d’une de ces variables exportées n’aura qu’une portée locale (pas de possibilité de retourner une valeur de cette manière).

On place une variable dans l’environnement exporté par la commande shell :

```
export NOM_DE_LA_VARIABLE
```

en SORTIE : **le code de retour :**

La variable shell **`$?`** contient le code de retour de la dernière commande lancée par le shell. Si la commande s’est terminée normalement, le code de retour vaut généralement 0 ; (le manuel de référence indique le code de retour de chaque commande).

Si la commande est elle-même un programme shell qui s’est terminé sur une instruction `exit` <valeur>, le code de retour vaut <valeur>, sinon il vaut le code de retour de la dernière commande exécutée dans le programme.

en ENTREE / SORTIE : **les fichiers.**

Par défaut, il y en a trois d’ouverts par le shell avant le lancement de chaque commande :

- 1 en entrée : l’entrée standard

- 2 en sortie : la sortie standard et la sortie d’erreur.

1. on appelle ici *argument* un champ placé après le nom de la commande sur la ligne de commande, et *paramètre de position*, cet *argument*, mais vu par la commande pendant son exécution.

Il est rappelée que le shell qui lance une commande positionne le nécessaire pour que ces entrées-sorties soient disponibles : mais, une fois lancée, il est du ressort de la commande de les utiliser et de retourner des valeurs couramment attendues[1]. I.3.2.

En général, une commande UNIX lit les données sur l'entrée standard, les modifie en fonction des arguments spécifiés, (options ou paramétrage) et renvoie le résultat sur la sortie standard. Si une erreur intervient, elle affiche un message sur la sortie d'erreur, et renvoie un code de retour différent de zéro.

Une bonne habitude consiste à respecter cette convention, ce qui permet ainsi d'écrire et d'intégrer facilement de nouvelles commandes au milieu de celles déjà existantes.

1. voir le chapitre consacré à ce sujet plus loin : *écrire un shell propre* page 187

I.5.1.

I.5. La manipulation des fichiers

I.5.1. Description

La manipulation d'un fichier passe toujours par les étapes suivantes :

1/ ouvrir le fichier

2/ écrire, lire dans le fichier

3/ fermer le fichier

Sous UNIX, on ne spécifie dans les commandes le nom du fichier physique utilisé que lors de l'étape 1 ; le système renvoie alors un numéro (un entier de 0 à 20) qui sert ensuite de référence logique pour toutes les autres actions liées à ce fichier : c'est le `descripteur.`

• Le descripteur 0 est nommé aussi :		entrée standard
•	1	sortie standard
•	2	sortie d'erreur

Lorsqu'une commande standard d'UNIX travaille, elle lit les données sur le descripteur 0, écrit le résultat sur le 1, et éventuellement les erreurs sur le 2, sans rien connaître en fait du nom des fichiers physiques qui lui sont associés.

Le shell, lui, ouvre en fonction de certaines directives (> < >> |, etc) les fichiers spécifiés en lecture ou en écriture, leur affecte à chacun un descripteur, lance la commande, attend la fin, puis ferme les fichiers associés aux descripteurs.

Par exemple, la commande `echo` écrit par convention sur le descripteur 1 ; mais ce résultat peut être dirigé vers n'importe quel fichier pourvu que le shell ait auparavant associé ce fichier au descripteur 1.

Une caractéristique importante des descripteurs est qu'ils font partie de l'héritage lors de la création d'un nouveau processus : un processus qui “naît” a des descripteurs qui pointent sur les mêmes fichiers que ceux de son père ; le shell de login (le premier lancé lors de votre connexion) affecte le descripteur 0 au clavier, et les descripteurs 1 et 2 à la console ; tant que vous ne modifierez pas la sortie standard (descripteur 1), tous les processus fils créés auront leur sortie standard associée à la console et leurs résultats s'y afficheront.

I.5.2. Exemple

commande : `grep truc <toto 2>trace`

Le shell qui évalue la ligne interprète :

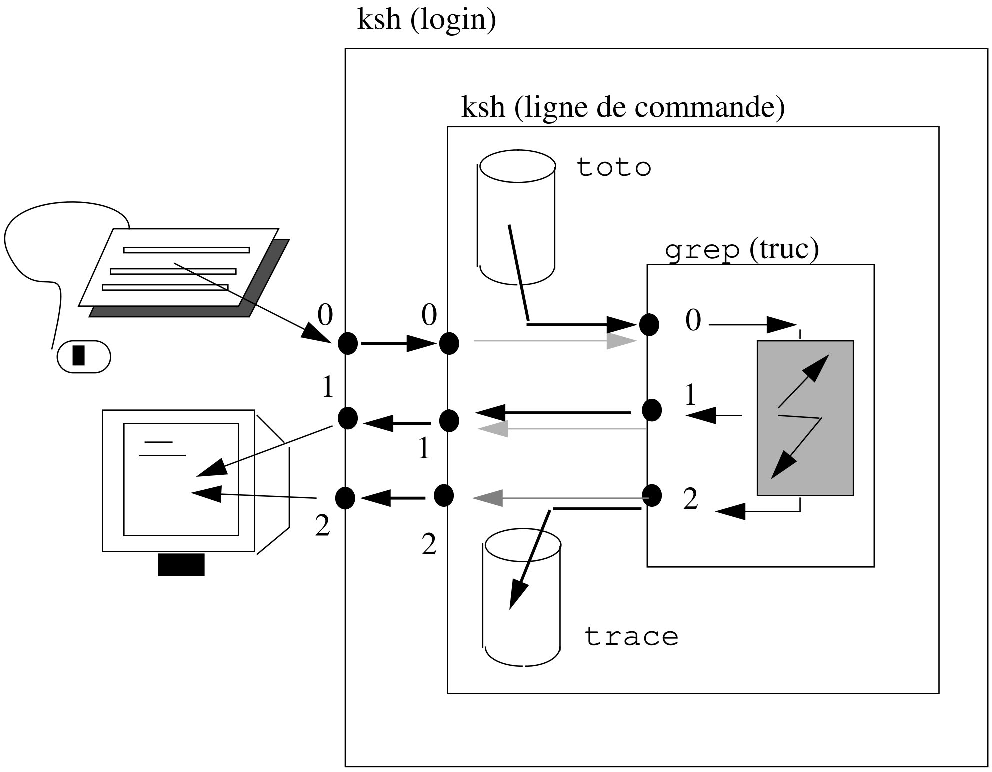
`<toto` par	: ouvrir le fichier `toto` en lecture et l'affecter au descripteur No 0 (entrée standard) du nouveau processus `grep`
`2>trace` par	: ouvrir le fichier `trace` en écriture et lui affecter le descripteur 2.

Comme aucune directive ne touche le descripteur 1, on ne le modifie pas. L'ancêtre du shell actuel, le shell de login par exemple, l'avait déjà affecté à la console : il y est encore par héritage. I.5.2.

Le nouveau processus `grep` lit les lignes présentes sur son entrée standard (descripteur 0), et recopie celles contenant la chaîne "truc" sur la sortie standard (descripteur 1).

Résultat : les lignes du fichier `toto` qui contiennent la chaîne "truc" s'affichent à l'écran, et le fichier `trace` est vide.

On peut suivre les connexions des différentes sorties de la commande sur le schéma suivant :



flèches grisées : l'état des connexions de la commande "grep" sans les redirections ;

flèches noires : avec les redirections.

I.5.3.

Remarque 1 : l'affectation des descripteurs (redirection) pour une commande ne touche pas le shell qui l'a effectuée ; on verra ensuite comment modifier les descripteurs à l'intérieur d'un shell.

Remarque 2 : Imaginons que la commande `grep` envoie ses messages d'erreur sur le descripteur 6 plutôt que sur le 2 comme le veut la convention, alors il aurait fallu écrire pour faire la même chose :

```
grep truc <toto 6>trace
```

Remarque 3 : Les redirections ne sont pas considérées comme des paramètres de la commande.

Dans le cas de :

```
grep truc <toto
```

le shell applique la redirection et lance la commande `grep` avec les deux paramètres "grep" et "truc".

Dans le cas de :

```
grep truc toto
```

le shell conserve les redirections déjà existantes et lance la commande `grep` avec les trois paramètres "grep", "truc" et "toto"[1].

En conclusion : les descripteurs sont des passerelles ;

le shell ouvre des fichiers et affecte les descripteurs ;

la commande lit et écrit sur les descripteurs (quelle que soit leur affectation).

Nous détaillerons tous les types de redirections plus loin.

I.5.3. Les droits et la manipulation des fichiers

Tous les processus et fichiers UNIX existant sur une machine peuvent être répartis dans trois ensembles différents :

- ceux appartenant à l'utilisateur donné U ;
- ceux appartenant à un autre utilisateur que U, mais faisant partie du même *groupe* G que U ;
- ceux n'appartenant ni à l'utilisateur U, ni à son groupe G : tous les autres.

Ces ensembles sont disjoints : il n'est pas possible qu'un processus ou un fichier appartiennent à plus d'un de ces ensembles.

Autant un utilisateur est défini de manière unique sur une machine par son numéro d'identification *(uid)*, autant le *groupe* de l'utilisateur *(gid)* est flottant : il est unique pour un

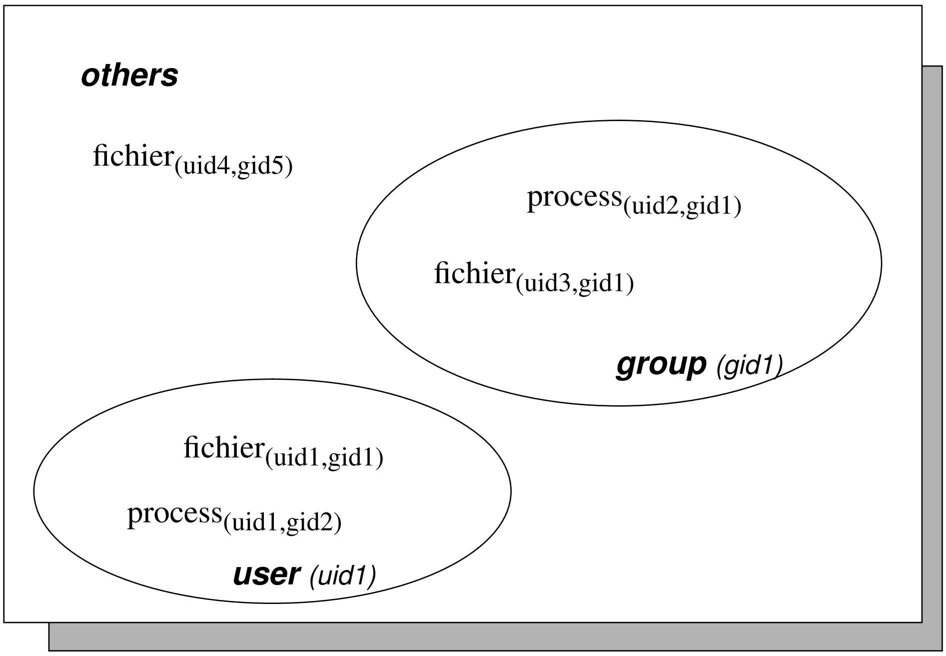
1. La commande `grep` doit se "débrouiller" seule pour savoir si elle doit lire les lignes de données sur le descripteur 0 (c'est ce qu'elle fait s' il n'y a que deux paramètres), ou si elle doit commencer par ouvrir le fichier dont le nom est le troisième paramètre (ce qu'elle fait uniquement s'il y a trois paramètres).

I.5.3.

processus ou un fichier donné, mais un utilisateur peut avoir de manière simultanée plusieurs processus ou fichiers ayant des groupes différents.

C'est l'administrateur du système qui définit les groupes dans lesquels vous pouvez vous positionner, ou positionner vos fichiers. Un processus ou un fichier est lié dans tous les cas à un doublé (uid,gid) :

- uid étant l'identificateur de l'utilisateur ayant créé le processus ou le fichier ;
- gid étant le groupe courant au moment de la création du processus ou du fichier.



Les trois ensembles ci-dessus définissent trois classes d'appartenance :

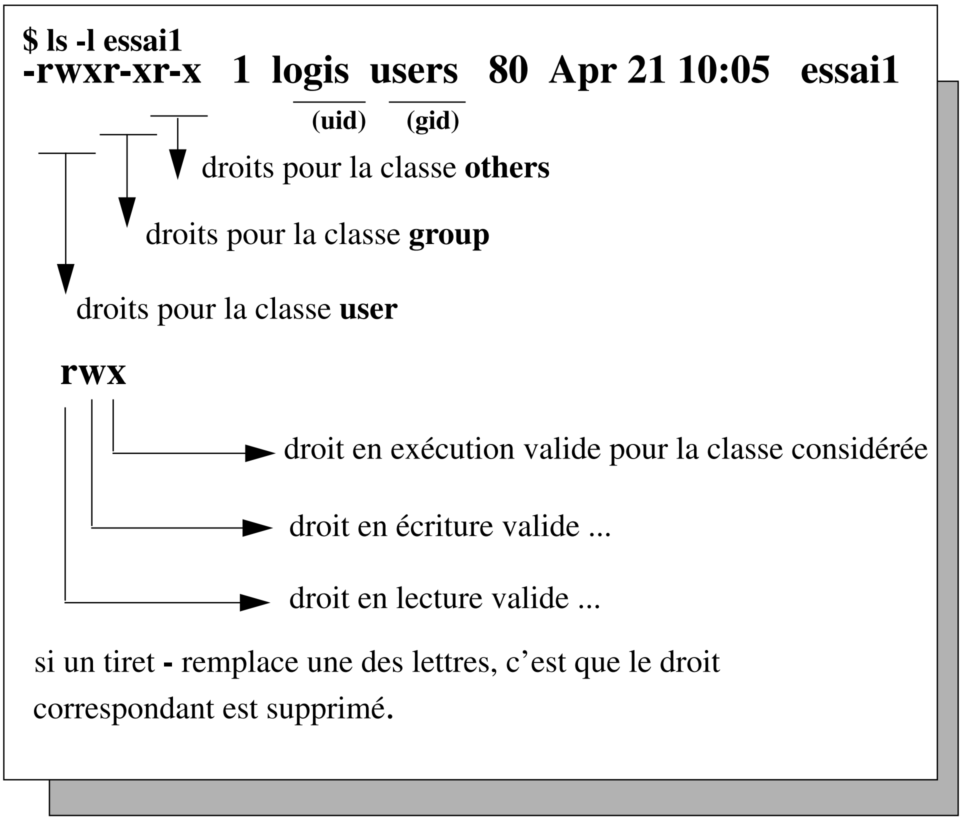
- *user* : la classe de tous les fichiers ou processus ayant le même propriétaire que l'utilisateur (ici, uid1)
- *group* : la classe des fichiers ou processus ayant le même groupe que l'utilisateur (ici, gid1).
- *others* : celle des fichiers ou processus n'ayant ni le même propriétaire, ni le même groupe que l'utilisateur (uid1).

On remarquera qu'il n'est possible d'appartenir qu'à une seule classe à la fois, et que la priorité va à la classe *user*. Ainsi, le propriétaire d'un fichier appartient d'office à la classe *user*, et par conséquent ne peut jamais appartenir ni à la classe *group*, ni à la classe *others*.

On peut définir sous UNIX des droits d'un fichier en lecture (r), en écriture (w), et en exécution (x) pour chacune de ces classes. Seul le propriétaire d'un fichier (ou l'administrateur[1]) peut modifier les droits de celui-ci .

1. Voir le paramétrage possible avec la commande `umask`: page 176

I.5.3. **<u>Symbologie :</u>**



Dans l'exemple du fichier `essai1` qui est indiqué ici :

- la classe **user** est en écriture, en lecture et en exécution : c'est à dire que le propriétaire peut lire, écrire et faire exécuter le fichier.
- la classe **group** a les droits en lecture et en exécution : c'est à dire que les utilisateurs ou les processus autres que le propriétaire et dont le groupe est identique à celui du fichier pourront lire et exécuter le fichier.
- la classe **others** a les droits en lecture et en exécution : les utilisateurs autres que le propriétaire du fichier et dont le groupe est différent de celui du fichier pourront le lire et l'exécuter.

On a indiqué plus haut qu'un utilisateur ou un processus ne pouvait appartenir qu'à une seule classe, cela signifie qu'il n'est concerné que par les droits de la classe qui lui correspond, les autres (pour les deux autres classes) étant non significatifs.

Pour connaître les droits que l'on possède sur un fichier, il suffit de comparer ses propres *`uid`* et *`gid`* (que l'on peut avoir en tapant la commande `id`) avec ceux du fichier :

```
$ id
uid=207(logis) gid=20(users)
$
```

- si mon `uid` est celui du fichier : je suis le propriétaire et les droits qui me concernent sont ceux de la classe *user*.

- si mon `uid` n'est pas celui du fichier, mais mon `gid` est celui du fichier, mes droits sont ceux de la classe *group*. I.5.3.

- si mon `uid` n'est pas celui du fichier, pas plus que mon `gid`, alors mes droits sont ceux de la classe *others*.

Droits sur les répertoires.

Pour l'instant, il n'a pas été indiqué comment on pouvait autoriser la création ou la suppression d'un fichier.

Lorsqu'on crée un fichier dans un répertoire, ou qu'on le détruit, cela revient à modifier le contenu du répertoire correspondant : on rajoute ou on enlève un élément dans la liste. Pour modifier le répertoire, il faut avoir les droits en écriture dessus.

Le fonctionnement des droits en lecture et en écriture sur un répertoire se fait de la même manière que pour les fichiers, en gardant à l'esprit que :

- la lecture d'un répertoire se fait lorsqu'on liste son contenu (par la commande `ls` par exemple, ou si on utilise des caractères spéciaux symbolisant les noms de fichiers : * ? [ ] ...)

  Même s'il n'est pas possible de lister le contenu d'un répertoire, on peut néanmoins atteindre et lister les fichiers des sous-répertoires, s'il en existe, et si on connaît leur nom.
- l'écriture dans un répertoire se fait lorsqu'on crée ou détruit un fichier
- l'exécution se fait lorsqu'un chemin de fichier ou une commande `cd` spécifie le nom de ce répertoire.

  De manière imagée, on pourrait penser que l'on “passe” par un répertoire lorsqu'on essaie d'atteindre un fichier que ce répertoire contient, ou qu'un sous-répertoire contient ; si l'exécution est interdite, on ne peut plus “passer”.

  En conclusion, si les droits en exécution sont enlevés sur un répertoire, il ne sera plus possible pour les utilisateurs de la classe correspondante d'utiliser des chemins qui contiennent ce répertoire.

En résumé :

Pour créer ou supprimer un fichier, il faut avoir les droits suffisant sur le répertoire qui le contient ;

Pour écrire, lire ou exécuter un fichier, il faut avoir des droits suffisants sur le fichier lui-même.

Remarque :

Lorsque vous créez un fichier de commande, pensez à rajouter les droits d'exécution dessus sans quoi vous ne pourrez pas le lancer ; vous obtiendrez le message : `cannot execute`.

I.5.4.

I.5.4. Les liens

Il arrive parfois d'avoir besoin, dans plusieurs répertoires, ou sous différents noms, de plusieurs exemplaires d'un fichier de référence.

Une première solution consiste à recopier physiquement le fichier aux différents endroits ou sous les différents noms.

Mais cela a deux inconvénients majeurs :

- d'une part, si ce fichier est commun à plusieurs utilisateurs ou applications, il est indispensable, chaque fois qu'une modification s'impose, de retoucher successivement les N copies déjà faites ;
- d'autre part, la place occupée au total est proportionnelle au nombre d'exemplaires nécessaires.

La solution consiste à faire des *liens* plutôt que des copies : il existe alors un seul exemplaire physique du corps du fichier (contenant les données), et plusieurs références pointant toutes vers cet unique corps.

La création d'un lien se fait avec une syntaxe identique à une copie classique :

pour une copie :

```
cp ancien nouveau
```

pour un lien :

```
ln ancien nouveau
```

ou bien `ln -s ancien nouveau`

Il y a deux façons de créer un lien, car il existe en fait deux types de liens :

- les liens *physiques*
- les liens *symboliques*

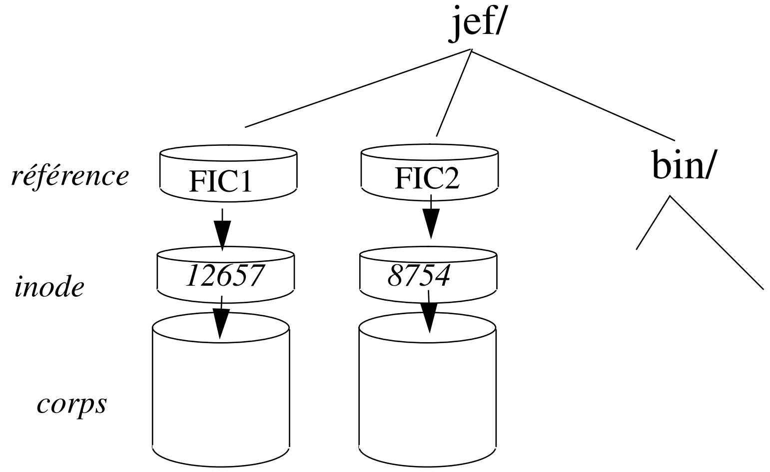
1.5.4.1. Les liens physiques

Pour comprendre ce qu'est un lien physique, il faut savoir qu'un fichier UNIX est composé de deux parties principales :

• le corps, qui contient les données proprement dites;

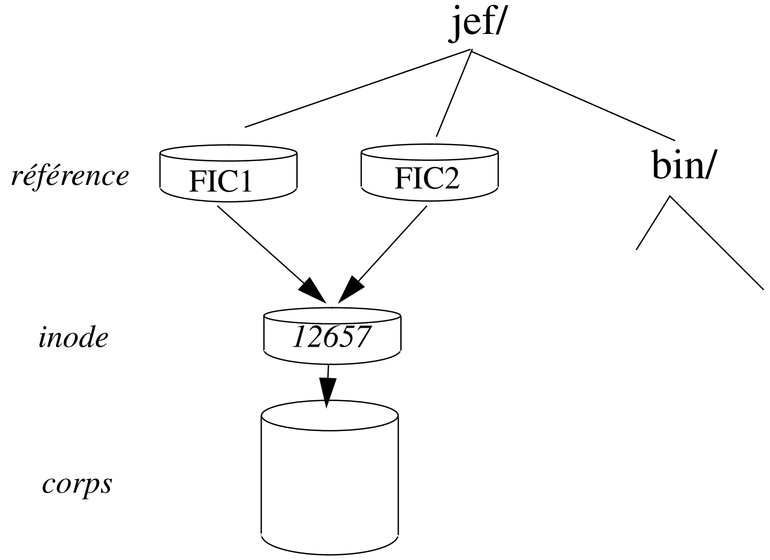
I.5.4.

- l'inode, qui contient les informations sur le fichier lui-même : droits, propriétaire, type, date de modification, etc... Voilà ce qu'on peut imaginer trouver dans le répertoire `jef` après avoir lancé la commande `cp fic1 fic2`, à partir d'un fichier `fic1` déjà existant.



Créer un lien physique, c'est créer une deuxième référence, dans le même répertoire, ou dans un autre, pointant sur une inode d' un fichier déjà existant.

Voilà le résultat de la commande `ln fic1 fic2`, à partir du fichier `fic1` déjà existant :



Dans le cas ci-dessus, les deux références jouent exactement le même rôle, la première n'ayant aucune caractéristique supplémentaire par rapport à la seconde (c'est encore vrai s'il y N liens sur un fichier, au lieu de deux comme ici).

création du lien

On vient de voir que la commande qui permet de créer un lien physique, appelé `nouveau`, et pointant sur le fichier `ancien` déjà existant est :

```
ln ancien nouveau
```

I.5.4. Exemple :

```
$ date >ancien
$ ls -l *
-rw-r--r-- 1 logis users 32 Jul 3 15:26 ancien
$ ln ancien nouveau
$ ls -l *
-rw-r--r-- 2 logis users 32 Jul 3 15:26 ancien
-rw-r--r-- 2 logis users 32 Jul 3 15:26 nouveau
$ echo >nouveau
$ ls -l *
-rw-r--r-- 2 logis users 1 Jul 3 15:27 ancien
-rw-r--r-- 2 logis users 1 Jul 3 15:27 nouveau
```

nombre de liens sur le fichier

Vous remarquerez que les deux fichiers, une fois le lien fait, ont tout en commun, sauf la référence (le nom) : toute modification apportée à un des deux fichiers est reportée sur l'autre. Il n'est pas possible par exemple, que ces deux fichiers liés aient des propriétaires différents, ou des droits différents.

La commande `ls -l` indique le nombre de liens physiques qui pointent sur le corps d'un fichier donné (voir l'exemple ci-dessus). Avant la commande de création du lien, le fichier `ancien` possédait un lien sur son corps. Après, on voit qu'il existe deux liens sur le corps duquel pointent les deux fichiers `ancien` et `nouveau`.

Remarque: il est possible de lier des fichiers depuis des répertoires différents. Il existe cependant des restrictions qui sont indiquées un peu plus loin.

suppression du lien

```
$ ls -l *
-rw-r--r-- 2 logis users 1 Jul 3 15:27 ancien
-rw-r--r-- 2 logis users 1 Jul 3 15:27 nouveau
$ rm ancien
$ ls -l *
-rw-r--r-- 1 logis users 1 Jul 3 15:27 nouveau
$
```

Chaque référence ayant un rôle équivalent, on peut supprimer n'importe laquelle (n'importe lequel des fichiers liés) sans toucher aux autres. Ce n'est que lorsqu'on supprime la dernière référence que le corps du fichier est supprimé aussi.

La suppression d'un lien physique se fait comme pour un fichier normal, avec la commande `rm`.

I.5.4.

petit exercice distrayant

Déterminer à partir du résultat de la commande `ls -l`[1] donné ci-dessous le nombre de sous-répertoires présents dans le répertoire `manuel_shell` :

```
$ ls -ld manuel_shell
drwxr-xr-x 4 logis users 1024 Apr 24 12:15 manuel_shell
$
```

correction (pour les feignants)

Les répertoires constituent un arbre ; ils sont chaînés les uns aux autres par l'intermédiaire des deux sous-répertoires : **.** et **..**, que l'on retrouve dans tout répertoire, même vide. Ces deux répertoires génériques sont en fait des liens : **..** est un lien sur le répertoire de niveau supérieur (`cd ..` permet de remonter d'un cran vers la racine /), et **.** un lien sur le répertoire courant.

A la création d'un répertoire (par `mkdir` par exemple), celui-ci est lié deux fois. Or chaque fois que l'on rajoute un sous-répertoire, on rajoute par la même occasion un lien **..** sur le répertoire courant. Donc, on a la relation suivante :

nombre de sous-répertoires = nombre de liens sur le répertoire courant - 2

Ici, on voit que le nombre de liens est 4 ; donc le nombre de sous-répertoires doit être 2. :

```
$ ls -l manuel_shell
total 6
drwxr-xr-x 3 logis users 2048 Jul  3 13:54 exemples
drwxr-xr-x 2 logis users 1024 Apr 24 12:48 perf
$
```

gagné !

Attention :

Il n'est pas possible de faire des liens physiques sur des répertoires... c'est une manipulation qui est réservée au système pour gérer les arborescences, mais qui n'est pas accessible aux utilisateurs.

1. la commande `ls -l` *`répertoire`* donne des informations sur le contenu de *`répertoire`*, alors que la commande `ls -ld` *`répertoire`* donne les informations sur le répertoire (ou à défaut le fichier) lui-même.

I.5.4. **Inconvénients d'un lien physique**

• La référence,l'inode et le corps d'un fichier sont des entités qui sont toujours réunies ensemble sur la même unité de disque, voire la même partition de disque. Or, la totalité de l'arborescence logique que vous pouvez accéder est le plus souvent découpée en plusieurs sous-arborescences, liées chacune à une unité de disque, ou une partition. Il n'est donc pas possible de lier des fichiers en les localisant dans deux répertoires appartenant à des disques ou partitions différents.

voir IBM

```
$ ln ancien /tmp/nouveau
ln: different file system
$
```

On a essayé de lier le fichier `ancien` qui se trouve sur le disque utilisateur, avec un que l'on voudrait placer dans `/tmp` qui se trouve (ici) sur le disque système : impossible.

• Il n'est pas possible de faire un lien physique sur un répertoire.

Avantage

• Si vous déplacez par la commande `mv` un des fichiers liés, (dans la limite du disque ou de la partition) tous les liens restent intacts.

1.5.4.2. Les liens symboliques

Plutôt que de dupliquer la référence, la création du lien symbolique va consister à créer un tout petit fichier d'un type spécial, et dont le contenu (la partie données) correspond à la localisation (absolue ou relative) du fichier auquel il est lié.

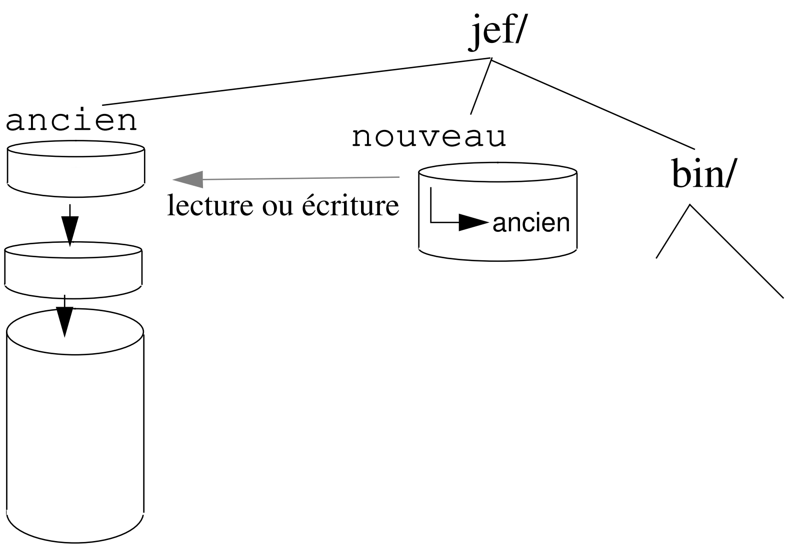
création du lien

La commande de création du lien symbolique *`nouveau`* pointant sur le fichier *`ancien`* est :

`ln -s` *`ancien nouveau`*

```
$ ls -l
total 2
-rw-r--r-- 1 logis users 1 Jul 3 15:27 ancien
$ ln -s ancien nouveau
$ ls -l
total 4
-rw-r--r-- 1 logis users 1 Jul 3 15:27 ancien
lrwxr-xr-x 1 logis users 6 Jul 3 17:35 nouveau -> ancien
$
```

Ici, on reconnaît le lien symbolique au caractère `l` qui précède les droits, et au nom du fichier sur lequel il pointe qui indiqué en fin de ligne. I.5.4.



voir
voir

Tant qu'on n'essaie pas de lire son contenu, un lien symbolique se comporte à peu près comme un fichier standard : on peut par exemple modifier son propriétaire. Par contre, dès qu'on tente d'y lire ou écrire, ou de changer ses droits d'accès, on accède en fait au données, ou à l'inode du fichier dont il a enregistré le nom (ou le chemin, s'il pointe sur un fichier d'un autre répertoire).

Un accès au contenu d'un lien symbolique nécessite pour le système deux accès successifs :

- le premier, pour lire sur quel fichier pointe le lien ;
- le second, pour accéder au fichier pointé, et lire ses données.

Remarquez donc que pour lire le contenu de `nouveau`, il faut avoir les droits en lecture dessus (pour pouvoir lire le chemin vers lequel il pointe), puis ensuite les droits sur le fichier pointé (c'est à dire `ancien`, pour pouvoir lire les données qu'il contient).

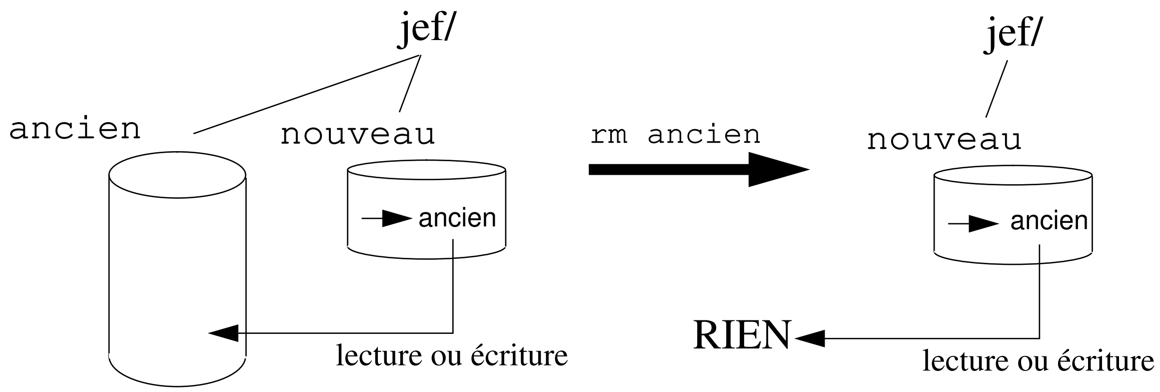
suppression du lien

Le fichier initial (qui contient réellement les données) n'est absolument pas touché de quelque manière que ce soit lorsqu'on déclare un lien symbolique pointant sur lui.

Ce qui veut dire que si le fichier initial est détruit ou déplacé, alors qu'il existait un lien symbolique pointant sur lui, le lien pointe maintenant sur RIEN !

Par contre, si on lance la commande `rm` sur le lien symbolique, seul le lien est perdu ; le fichier sur lequel il pointait n'est pas concerné.

I.5.4. Une tentative de lecture d’un lien symbolique pointant sur un fichier inexistant provoque le message : `No such file or directory`



Inconvénients

• Un lien symbolique augmente le temps d’accès au données : le système de fichiers doit faire une première lecture dans le lien pour connaître le chemin du fichier contenant les données ; puis un deuxième accès vers le fichier de données lui-même.

• Si le fichier sur lequel pointe le lien symbolique est déplacé ou supprimé, le lien continue de pointer vers un fichier inexistant : il faut le détruire et le reconstruire.

Avantages

• Le problème de disque physique ou de partition n’apparait pas ici ; on peut faire un lien symbolique sur tout fichier de l’arborescence.

• Il est possible de faire des liens symboliques sur des répertoires

• la commande `ls -l` permet de savoir clairement vers quel fichier pointe le lien.

• Si le fichier ancien est supprimé puis recréé ensuite sous le même nom au même endroit, (par une procédure automatique, ou lors d’une montée de niveau logicielle par exemple), le lien est toujours valide.

Pour les deux types de liens, vous trouverez dans le chapitre sur les cas types des exemples de commandes permettant de lister tous les liens qui pointent vers un fichier donné : voir page 174

I.6. Interprétation d'une commande shell

I.6.1.

Une ligne shell comprend généralement un certain nombre de symboles et de caractères spéciaux. L'interpréteur va les évaluer, puis les remplacer par les chaînes de caractères correspondantes, en fonction de certaines règles que nous allons détailler.

Pour bien évaluer ce que votre ligne va devenir une fois interprétée, il est bon d'imaginer la démarche du shell lorsqu'il manipule votre ligne.

On peut considérer que la transformation se fait par passes, en s'effectuant dans l'ordre qui suit :

1	Interpréter les séparateurs et séparer les champs
2	Isoler les chaînes de caractères encadrées par des 'quotes'
3.1	Remplacer les variables par leur contenu
3.2	Exécuter les commandes encadrées par des "anti-quotes" ou la séquence $() et les remplacer par leur résultat
4	Isoler les chaînes de caractères encadrées par des 'double-quotes'
5	Remplacer les caractères spéciaux (*, [], ?, ~ , etc...) par leur valeur
6	Repérer les séquences () ou { }
7	Positionner certains paramètres en fonction des caractères spéciaux ; | & && ||
8	Mettre en place les redirections
9	Localiser les commandes élémentaires (alias, fonctions, path)

L'étape 3 est décomposée en deux sous-niveaux équivalents car les résultats respectifs de ces deux sous-niveaux ne peuvent pas être utilisés l'un par l'autre (en quelque sorte, les deux transformations sont faites en parallèle).

Evidemment, il existe des cas particuliers qui n'améliorent pas vraiment la simplicité de l'algorithme donnée ci-dessus ; vous pouvez toujours vous reporter au manuel des commandes UNIX (section 1) sur le ksh, section "Specials commands", pour les détailler.

I.6.1. Etape 1 : Interprétation des séparateurs

Comme dans le cas d'une commande simple, une commande shell est découpée en paquets qui seront évalués un à un. Couramment, les séparateurs de champs (ou paquets) sont les caractères <espace> et <tabulation>. En fait, ces séparateurs sont placés dans la variable IFS de l'environnement, et rien n'empêche de modifier son contenu pour que le shell travaille avec d'autres séparateurs.

En fin d'évaluation de la ligne, tout ce qui est accolé sans séparateur forme un champ ; ainsi pour concaténer plusieurs chaînes, il suffit de les mettre bout à bout.

I.6.2.

I.6.2. Etape 2 : Le caractère spécial quote '

Toute chaîne encadrée par des caractères quote est prise telle quelle et n'est pas transformée. Aucun des symboles du shell ni des caractères spéciaux n'est évalué.La chaîne contenue (y compris les espaces, tabulations, etc..) constituera un champ de la commande qui sera lancée. Ceci explique l'exemple suivant :

```
$ ls
toto1.c toto2.c toto_base.c tutu
$ ls 'toto1.c toto2.c'
toto1.c toto2.c  not found
$ ls toto1.c toto2.c
toto1.c toto2.c
```

Lorsque les noms des deux fichiers sont placés entre quotes, l'ensemble, y compris l'espace entre les deux noms forme un seul champ, qui est donc évalué par la commande UNIX `ls` ; or, il n'existe pas dans le répertoire de fichier dont le nom est : "toto1.c<espace>toto2.c" ; la commande échoue.

Attention : Il n'est pas possible de placer un caractère quote à l'intérieur d'une chaîne elle-même définie par des quotes.

Pour faire perdre au caractère quote sa signification particulière, on peut le "protéger" par le caractère \, ou bien le placer lui même à l'intérieur d'une chaîne encadrée par des double-quotes (!!).

I.6.3. Etape 3 : Les variables et le caractère spécial anti-quote ou $()

1.6.3.1. Etape 3.1 : Les variables

Les variables du shell sont composées de lettres de l'alphabet, majuscules ou minuscules (plus le caractère souligné : _), et de chiffres, sauf en première position ; par défaut, elles ont toutes le type : chaîne de caractères.

Si le nom de la variable n'est constitué que de chiffres, son contenu est un des paramètres fourni au programme en cours lors de son démarrage. ($0 est le nom du programme en cours, $1 le premier argument fourni à l'appel du programme en cours, $3 le troisième et ${10}[1] le dixième : voir les variables prépositionnées page 38.

1. $10 n'est pas le contenu du 10ème paramètre, mais le contenu du premier paramètre suivi d'un zéro. Voir plus loin la signification des { } dans les noms de variable.

Pour affecter une variable, on la fait suivre, sans séparateur, du signe égal et d'un champ (la chaîne voulue). I.6.3.

Pour utiliser une variable (la lire), on fait précéder son nom du caractère $

```
$ toto=un
$ tutu='deux trois'
$ echo $toto $tutu toto
un deux trois toto
```

On voit ci-dessus que le champ avec lequel on affecte la variable peut être composé grâce aux “quotes”, ou de la même façon avec des double-quotes, qu'on décrit plus loin page 42.

Pour concaténer le contenu de plusieurs variables, il suffit de les accoler entre elles ; de la même manière, pour concaténer le contenu d'une variable et une chaîne fixe, il suffit de les accoler.

```
$ s1=Korn
$ s2=Shell
$ echo $s1$s2
KornShell
$ echo Korn$s2
KornShell
$ echo $s1-$s2
Korn-Shell
```

Mais dans ce cas, il peut exister des ambiguïtés sur le nom de la variable (où finit le nom de la variable et où commence la chaîne fixe?), sauf si la chaîne fixe commence par un caractère autre que ceux autorisés pour les noms de variables.

I.6.3. Pour lever l'ambiguïté, si nécessaire, on encadre alors le nom de la variable par des accolades { }, toujours précédées du caractère $.

```
$ s1=Korn
$ echo $s1Shell

$ echo ${s1}Shell
KornShell
```

Dans l'exemple ci-dessus, la commande `echo $s1Shell` ne donne rien car le shell essaie de fournir le contenu de la variable dont le nom est `s1Shell,` qui n'existe pas.

Une commande du shell (la commande `set`) permet de choisir si une tentative de lecture d'une variable inexistante doit provoquer un message d'erreur ou non ; ici, il n'y a pas de message d'erreur. Cela peut être pratique, mais aussi la cause de disfonctionnement apparemment incompréhensible, si une faute de frappe s'est glissée dans votre programme (oubli du caractère $ devant un nom de variable !).

Environnement exporté

Pour qu'une variable soit placée dans l'environnement exporté[1], on utilise la commande export :

`export` *`nom_de_variable`*

On peut réaliser l'affectation en même temps :

`export` *`nom_de_variable=valeur`*

Tableaux

Il est possible de définir des tableaux à une dimension de la manière suivante :

nom_du_tableau[*indice1*]=<champ1>

nom_du_tableau[*indice2*]=<champ2>

etc..

ou de cette façon :

`set +A` *`nom_du_tableau`* `<champ1> <champ2> ...`

auquel cas le tableau *`nom_du_tableau`* est initialisé dans l'ordre avec les paramètres fournis. (voir la commande `set` au chapitre qui est consacré aux commandes du shell)

1. c'est à dire l'environnement vu automatiquement par tout processus au moment de sa naissance

Pour lire un élément du tableau `toto`, on utilise la syntaxe suivante : I.6.3.

`${toto[<indice>]}`

```
$ toto=ksh
$ toto[1]=Korn
$ toto[2]=Shell
$ i=1
$ echo ${toto[$i]}
Korn
$ echo $toto[1]
ksh[1]
```

La dernière commande montre bien que la syntaxe du shell est parfois peu orthodoxe ! (`$toto[1]` n'est pas considéré comme le premier élément du tableau `toto`, mais comme le contenu de la variable toto accolé avec la chaîne littérale `[1]`)

Il n'est pas possible d'exporter des tableaux entiers : vous n'arriverez qu'à exporter le premier élément du tableau en utilisant la commande export.

Variables prépositionnées

Lorsque le shell courant ou un programme en shell démarre, son environnement contient un certain nombre de variables qui sont positionnées dynamiquement, et dont le contenu, pour les principales, est le suivant :

$0	le nom du programme shell en cours.
$1, ..${N}	les N paramètres passés au programme (au shell) lors de son appel.
$#	le nombre de paramètres passés à l'appel du programme shell (non compris le paramètre $0)
$*	la liste des paramètres passés à l'appel du programme shell (non compris le paramètre $0)
$$	le numéro de processus courant (il y a un numéro unique par processus sur la machine)
$PPID	le numéro du processus parent du processus courant (de ce shell)
$RANDOM	un nombre aléatoire, entre 0 et 32767 (change à chaque lecture)
$IFS	la liste des séparateurs de champs dans une ligne du shell (par défaut <espace> et <tabulation>)
$HOME	votre répertoire principal
$PWD	le répertoire courant
$PATH[1]	la liste des répertoires où le shell est susceptible de trouver les commandes élémentaires à exécuter. Cette variable est toujours positionnée dans les

1. voir aussi le paragraphe sur le path page 58

I.6.3.

fichiers profile (`/etc/profile` et `.profile`) au moment du démarrage, mais n'est pas, en fait, modifiée dynamiquement par la suite.

$SECONDS le contenu initial de la variable SECONDS plus le nombre de secondes écoulées depuis sa dernière affectation. Si la variable n'a jamais été affectée, elle n'apparait pas.

$LINENO le numéro de la ligne dans le programme ou la fonction en cours

$PS1 votre prompt (ce qui s'affiche après chaque ligne de commande en mode intéractif)

$? le code d'erreur de la dernière commande exécutée.

<u>Opérateurs sur les variables.</u>

Il existe de nombreux opérateurs que l'on peut appliquer lorsqu'on lit une variable. Voici les plus courant :

${#toto} la longueur du contenu de la variable toto

(Par la suite, on désignera par <P> un "pattern",ou expression générique[1], c'est à dire une suite de caractères éventuellement spéciaux qui représente un ensemble de chaînes correspondantes. Par exemple, `toto*` est un "pattern", qui représente toutes les chaînes de caractères commençant par `toto` et suivies par n'importe quoi.)

${toto#<P>} le contenu de la variable toto sans son début si celui-ci correspond au pattern.

${toto##<P>} la même chose que précédemment, mais la plus grande partie possible est supprimée.

${toto%<P>} le contenu de la variable toto sans sa fin si celle-ci correspond au pattern

1. Voir page 70

I.6.3.

${toto%%<P>} la même chose que précédemment, mais la plus grande partie possible est supprimée.

```
$ toto=/users/petrus/logis/truc.def
$ echo ${#toto}
28
$ echo ${toto#*/}
users/petrus/logis/truc.def
$ echo ${toto##*/}
truc.def
$ echo ${toto%.def}
/users/petrus/logis/truc
$ echo ${toto%.*}
/users/petrus/logis/truc
$ echo ${toto%pouet}
/users/petrus/logis/truc.def
$ echo ${toto%ru*}
/users/petrus/logis/t
$ echo ${toto%%ru*}
/users/pet
```

Dans l’exemple ci-dessus, on utilise souvent pour former le “pattern” le caractère * , qui est un caractère spécial signifiant : < une chaîne comprenant n’importe quel nombre de n’importe quel caractère[1]>.

De plus, il est possible de placer le pattern dans une variable, et de décomposer le pattern en plusieurs éléments::

```
$ toto=/users/petrus/logis/truc.def
$ pattern='ru*'
$ echo ${toto%%$pattern}
/users/pet
$ pattern=ru
$ expand='*'
$ echo ${toto%%${pattern}${expand}}
/users/pet
$
```

1. Voir page 70

I.6.3.

et, toujours plus fort, d'imbriquer les substitutions:

```
$ toto=/users/petrus/logis/truc.def
$ pattern1=rubis
$ pattern2=bis
$ expand='*'
$ echo
${toto%%${pattern1%$pattern2}$expand}
/users/pet
$
```

1.6.3.2. Etape 3.2 : Le caractère spécial anti-quote ‘ ou $( )

Lorsqu'une chaîne est encadrée par des anti-quotes ou par la séquence $( ), elle est considérée comme une commande, et la séquence est remplacée par son résultat (sortie standard de la commande) une fois exécutée.

Cependant, les étapes vues au-dessus ne sont pas réévaluées : la chaîne est exécutée telle quelle à ce niveau d'interprétation, en fonction des étapes 4 et suivantes.

```
$ ls
toto1.c toto2.c toto_base.c tutu
$ echo $(ls) $(echo '$PWD')
toto1.c toto2.c toto_base.c tutu $PWD
```

Si on décompose l'interprétation de la ligne donnée en exemple ci-dessus,

- la chaîne $PWD est encadrée par des quotes, elle est donc protégée (étape 2) ;
- $(ls) est remplacé par le résultat de la commande `ls`, (étape 3) soit :

  `toto1.c toto2.c toto_base.c tutu`

- $(`echo '$PWD'`) est remplacé (étape 3) par le résultat de la commande `echo` sur la chaîne littérale $PWD (puisque la chaîne $PWD a été protégée) ;

A la fin de cette étape, la ligne se présente donc sous la forme suivante :

`echo toto1.c toto2.c toto_base.c tutu $PWD`

Comme l'étape de transformation des variables n'est faite qu'une fois, $PWD n'est pas évaluée à ce niveau (nous sommes à l'étape 4 ou plus), et on obtient le résultat indiqué.

Il est possible de faire des redirections, pour récupérer un message d'erreur, par exemple : I.6.4.

```
$ ls
toto1.c toto2.c toto_base.c tutu
$ res=$(ls truc) ; echo $res
truc not found

$ res=$(ls truc 2>&1) ; echo $res
truc not found
$
```

Si on décompose l'interprétation de la première ligne donnée en exemple ci-dessus,

- la variable `res` est affectée avec la sortie standard de la commande `ls truc`, c'est à dire rien puisque le fichier truc n'existe pas dans le répertoire courant. La ligne vide qui apparait est le résultat de la commande `echo $res`, puisque la variable `res` est vide.

On remarque cependant que le message "truc not found" apparait : cela vient du fait que la sortie d'erreur de la commande `ls truc` est héritée du shell, c'est donc la console.

Si on décompose la deuxième ligne donnée en exemple, on constate que la seule différence vient de la redirection : la variable `res` est affectée avec la sortie standard de la commande `ls truc`, mais aussi avec sa sortie d'erreur ; le message d'erreur "`truc not found`" n'est donc plus envoyé vers la console, mais dans la variable `res` (qui est ensuite affichée).

I.6.4. Etape 4 : Le caractère spécial double-quote "

La chaîne de caractères comprise entre deux double-quotes formera un champ de la commande élémentaire qui sera lancée, de la même façon que pour les chaînes encadrées par des quotes simples. La différence avec le caractère simple quote est qu'à cette étape, les transformations des symboles de variables, et séquence de $() ont déjà été évaluées. Par contre, les caractères spéciaux * ~ [] etc.. (voir étape 5) perdent leur signification s'ils sont placés entre double-quotes.

On encadrera donc de double-quotes des chaînes qui contiennent des variables, `$()`, etc... que le shell doit évaluer, et de simples quotes les chaînes qui ne doivent pas être évaluées.

Les chaînes encadrées de double-quotes peuvent être concaténées entre elles ou avec une autre chaîne en les accolant ; même si elles contiennent des séparateurs, elles seront considérées comme un champ unique lors de l'appel final de la commande élémentaire.

I.6.5. Un caractère quote à l'intérieur d'une chaîne encadrée par des doubles-quotes perd sa signification spéciale ; l'inverse est vrai également. Pour placer le caractère littéral (passif) double-quote dans une chaîne elle-même encadrée par des double-quotes, on peut le protéger par un ****.

```
$ echo "'"
'
$ echo '"'
"
$ echo "$PWD $(pwd) '$PWD'"
/tmp /tmp '/tmp'
$ echo "~ *"
~ *
$ echo ~ *
/users/logis toto1.c toto2.c tutu
$ echo " $RANDOM"===="12 34"'56'
 31079====12 3456
```

On peut vérifier sur les exemples ci-dessus le rôle des double-quotes. En particulier, le 3ème `echo` encadre son argument entre double-quotes : cela veut qu'à l'intérieur, le simple-quote perd sa signification, mais que les variables sont remplacées par leur valeur; c'est bien ce qui se passe pour le `$PWD` en bout de ligne.

I.6.5. Etape 5 : Les caractères spéciaux * [] ? ~ \

Lorsqu'on désigne un nom de fichier, il est souvent peu pratique d'indiquer le nom ou le chemin complet. Il existe des caractères spéciaux qui permettent de désigner symboliquement des caractères, ou des parties du nom de fichiers. Le shell recherche alors dans les répertoires désignés, ou par défaut dans le répertoire courant le ou les fichiers dont le nom est conforme à la description spécifiée. Attention : on imagine souvent que c'est la commande qui interprète ces caractères ; en fait, il sont transformés par le shell avant que la commande ne soit lancée.

Voici la correspondance :

caractère * : n'importe quelle partie ou totalité du nom d'un fichier ou de répertoire. Si plusieurs occurrences sont possibles, la chaîne contenant * (également nommée "pouet") est étendue autant de fois.

I.6.5.

Exemple :

```
$ ls
toto1.c toto2.c toto_base.c tutu
$ echo toto*.c
toto1.c toto2.c toto_base.c
```

caractère ? : un caractère quelconque.

Exemple :

```
$ ls
toto1.c toto2.c toto_base.c tutu
$ echo toto?.c
toto1.c toto2.c
```

séquence [ ] : un caractère parmi ceux spécifiés entre les crochets.

[12] : le caractère 1 ou 2

[1-5] : le caractère 1 ou 2 ou 3 ou 4 ou 5

[A-] : un caractère parmi la liste des caractères ascii de rang supérieur ou égal à ‘A’.

[!1-5] : un caractère parmi les caractères ascii qui ne font pas partie de l’ensemble des caractères de 1 à 5.

Exemple :

```
$ ls
toto1.c toto2.c toto_base.c tutu
$ echo toto[!_].c
toto1.c toto2.c
```

caractère ~ : le chemin absolu du répertoire courant au moment de votre login (“home directory”). Si ~ précède un nom d’utilisateur, alors cela représente le chemin absolu du “home directory” de l’utilisateur.

I.6.5. Exemple :

```
$ echo ~logis/.profile
/users/petrus/logis/.profile
```

caractère / : Le symbole **/** est le séparateur entre les différents noms de répertoires qui composent un chemin de fichier (un chemin absolu commence par **/**, alors qu'un chemin relatif débute dans le répertoire courant par un terme non précédé de **/**).

Attention : Créer un fichier dont le nom comprend un caractère spécial (un de ceux vu ci-dessus) n'amène en général que des ennuis : il est très difficile de désigner ce fichier par la suite.

caractère \ : Si vous voulez rendre inactif un des caractères spéciaux du shell, il faut le faire précéder du caractère ****.

la séquence ***** vaut ***** et non pas la liste des fichiers du répertoire courant.

la séquence **** vaut ****

la séquence **\'** vaut **'** et n'est pas le symbole du début d'une chaîne protégée.

Il est possible de composer les caractères ci-dessus pour obtenir des expressions plus complexes[1]. Si on note *expression* une suite de caractères, éventuellement spéciaux, alors :

?(*expression*)

?(*expression1*|*expression2*|...)

remplace zéro ou une occurrence des expressions fournies.

*(*expression*)

*(*expression1*|*expression2*|...)

remplace zéro ou plusieurs occurrences des expressions.

+(*expression*)

+(*expression1*|*expression2*|...)

remplace une ou plusieurs occurrences des expressions.

@(*expression*)

@(*expression1*|*expression2*|...)

remplace une occurrence exactement d'une des expressions fournies.

1. appelées aussi expressions génériques: voir page 70

!(*expression*)

!(*expression1*|*expression2*|...)

est remplacé par toutes les occurrences possibles sauf celles correspondant aux expressions fournies.

Exemples :

```
$ ls
essai1 essai1.c toto1.c toto_base.c
essai1.0 essai2 toto22.c tyty
$ echo !(toto*)
essai1 essai1.0 essai1.c essai2 tyty
$ echo +(to)*
toto1.c toto22.c toto_base.c
$ echo essai1?(.c)
essai1 essai1.c
$ echo *+([0-9]).c
essai1.c toto1.c toto22.c
$
```

Il est rappelé que les expressions sont évaluées en fonction des noms de fichiers qui existent dans le répertoire courant si l’expression ne contient pas de caractère /, et dans le répertoire indiqué sinon.

I.6.6. Etape 6 : les séquences () et {}

Il est possible de fabriquer une macro-commande à partir de plusieurs commandes élémentaires ; une macro-commande est encadrée par des () ou des {}, et se comporte comme une commande normale : par exemple, on peut rediriger ses entrées et ses sorties.

Si on désire que la macro-commande s’exécute dans l’environnement courant, on utilisera les {} ; si on préfère qu’elle s’éxécute dans un environnement séparé, on utilisera les () ; en pratique, c’est cette configuration qui est la plus intéressante.

I.6.7. Exemple :

```
$ pwd; (cd /usr/pub/; pwd ; ls); pwd
/tmp
/usr/pub
ascii eqnchar
/tmp
$ pwd; { cd /usr/pub/; pwd ; ls ; }; pwd
/tmp
/usr/pub
ascii eqnchar
/usr/pub
$
```

Vous constatez que le changement de répertoire est local pour une macro-commande encadrée par des (), alors qu'elle continue à garder son effet lorsqu'on utilise des {}.

De plus, la syntaxe de la macro-commande est un peu plus rigide si on utilise des {} : il faut isoler les { } entre des espaces et faire suivre chaque commande par un point virgule.

I.6.7. Etape 7 : Les caractères spéciaux ; | && & |

Ces caractères (ou séquences) définissent la fin d'une commande élémentaire. On peut donc composer une ligne shell par plusieurs commandes élémentaires séparées par un de ces caractères ; suivant le choix du caractère, l'intéraction entre les deux commandes contiguës sera différente.

1.6.7.1. caractère ;

Ce caractère peut être considéré comme un opérateur séquentiel : les deux commandes contiguës sont exécutées l'une après l'autre (la seconde démarre lorsque la première est terminée).

L'effet est le même si on place les deux commandes l'une après l'autre sur deux lignes consécutives du programme.

1.6.7.2. séquence ||

Dans le cas où deux commandes sont séparées par des ||, la deuxième n'est exécutée que si la première échoue, c'est à dire si son code de retour est différent de zéro.[1]

1. Des exemples sont visibles au chapitre sur les expressions conditionnelles. Voir page 71

1.6.7.3. séquence && I.6.7.

Dans le cas où deux commandes sont séparées par des **&&**, la deuxième n'est exécutée que si la première réussit, c'est à dire si son code de retour est égal à zéro[1].

1.6.7.4. caractère &

Ce caractère peut être considéré comme un opérateur asynchrone (parallèle) ; le shell n'attend pas la fin de la commande précédent le & pour lancer celle qui le suit. Par exemple, si la ligne de commande shell est de la forme :

```
cmd1 & cmd2 & cmd3
```

alors, la commande `cmd1` est exécutée, puis sans attendre la fin de `cmd1`, la commande `cmd2` est lancée, de même la commande `cmd3`. Lorsque la commande `cmd3` est finie, on passe à la ligne suivante du programme.

Autre exemple :

```
cmd1 & cmd2 &
```

la commande `cmd1` est lancée, puis sans attendre, la commande `cmd2` aussi ; et comme elle est suivie du &, le shell passe à l'interprétation de la ligne de programme suivante, sans attendre la fin de `cmd1` ni `cmd2`.

Le fait de lancer une commande de manière asynchrone avec un & nécessite de créer un processus indépendant ; ce que nous avons déjà vu sur l'héritage est valable ici aussi (en particulier pour les redirections) ; nous verrons qu'il est possible depuis un shell interactif de garder un certain contrôle sur ce types de processus, appelés aussi “jobs”.

Il arrive qu'on nomme “arrière plan” ou “background” la classe de processus lancés de manière asynchrone, en opposition avec “l'avant plan” ou “foreground” qui est la classe des processus s'exécutant séquenciellement dans un programme.

1.6.7.5. le caractère | ou “pipe”

Cet opérateur permet de lancer les deux commandes contiguës en reliant la sortie standard (descripteur 1) de la première commande dans l'entrée standard (descripteur 0) de la seconde. Le nom “pipe” donne l'idée d'un tuyau par lequel les données transitent d'une commande à l'autre.

Il est possible ainsi de chaîner autant de commandes que l'on désire, le seul impératif étant que chacune des commandes lise les données d'entrée sur le descripteur 0 et fournisse les résultats sur le descripteur 1 (éventuellement après une redirection).

Les deux ou N commandes reliées par des “pipes” sont démarrées simultanément dans autant de processus différents et disparaissent une à une lorsque leur entrée standard est vide. Le shell passe à

1. Voir les expressions conditionnelles: page 71

I.6.8. l'exécution de la commande ou de la ligne de programme suivante lorsque toutes les commandes liées par des pipes ont disparu.

```
$ ls
toto1.c toto2.c toto_base.c tutu
$ ls | grep toto | sort
toto1.c
toto2.c
toto_base.c
```

L'exemple ci-dessus est bien compliqué pour le résultat escompté, mais il montre comment il est possible de combiner plusieurs commande UNIX pour en constituer une plus puissante ou mieux adaptée au besoin courant.

voir

```
$ ls
toto1.c toto2.c toto_base.c tutu
$ ls toto tata | wc -l
toto not found
tata not found
 0
$ ls toto tata 2>&1 | wc -l
 2
```

Dans ce cas, on met à jour l'effet des redirections :

- la première ligne de l'exemple consiste à "piper" une commande qui génère des messages d'erreur dans une autre (`wc -l` compte les lignes de l'entrée standard). Comme les fichiers `toto` et `tata` n'existent pas dans le répertoire courant, la commande `ls` ne fournit rien sur la sortie standard, et deux messages apparaissent sur la sortie d'erreur, un pour le fichier `toto` et un pour le fichier `tata` ; ayant hérité des descripteurs du shell intéractif, ces messages sont envoyés sur la console et la commande `wc -l` indique un nombre nul de lignes provenant de l'entrée standard.
- la deuxième ligne est identique, à ceci près que, grâce à la redirection, les messages d'erreur sont redirigés dans l'entrée standard du "pipe" et que ces deux lignes sont bien comptées par la commande `wc -l`

Il est possible de "piper" entre elles des commandes UNIX, des commandes internes du shell, mais aussi des macro-commandes, comme elles ont été décrites au chapitre précédent.

I.6.8. Etape 8 : Les redirections

Nous avons déjà vu dans le chapitre sur les fichiers ce qu'étaient les redirections : une association entre un descripteur et un nom de fichier physique.

Voici les différentes manières de rediriger les sorties et les entrées d'une commande, sachant qu'on peut en combiner plusieurs sur une commande donnée : I.6.8.

digit>name : le descripteur de numéro `digit` est associé avec le fichier physique de nom `name` ouvert en écriture. Si le fichier n'existait pas auparavant, il est créé.

Si `digit` est omis, la valeur 1 est prise par défaut (sortie standard).

digit>>name : la même chose, mais le fichier est ouvert en mode *ajout* (on écrit à la fin du fichier existant).

digit<name : la même chose, mais cette fois le fichier `name` est ouvert en lecture ; il doit exister.

Si `digit` est omis, la valeur 0 est prise par défaut (entrée standard).

digit<<word : associe le descripteur `digit` avec le fichier du programme en cours, à partir de la ligne qui suit cette déclaration jusqu'à la ligne précédant celle débutant par le terme `word`.

Cela permet d'inclure dans des programmes des zones de texte à utiliser ou à recopier dans d'autres fichiers au moment de l'exécution. La commande redirigée de cette manière utilise comme entrée standard les lignes qui suivent immédiatement dans le programme, jusqu'au terme `word`.

d1>&d2 : le descripteur `d1` est associé en mode écriture avec le fichier déjà associé au descripteur `d2`.

Si `d1` est omis, la valeur 1 est prise par défaut ; `d2` ne peut pas être omis. Le fichier déjà associé au descripteur `d2` doit avoir été ouvert lui aussi en écriture.

d1>>&d2 : même chose mais `d1` est associé en mode ajout.

d1<&d2 : même chose, mais `d1` est associé en mode lecture ; si `d1` est omis, la valeur 0 est prise par défaut ; le fichier associé à `d2` doit avoir été ouvert en lecture.

Les redirections ci-dessus sont placées à la fin d'une commande et ne prennent effet que sur la commande qui précède juste. Pour modifier une fois pour toutes la redirection d'un descripteur à l'intérieur d'un programme shell, on utilise la syntaxe suivante :

```
exec <redirection>
```

la redirection <*redirection*> qui prend une des formes vues ci- dessus est appliquée au shell en cours et donc par héritage, à toutes les commandes que celui-ci lancera.

Si vous tapez par exemple `exec >/tmp/toto`, vous n'aurez plus sur votre console que le prompt et les messages d'erreur qui vont s'afficher ; le résultat de toutes vos commandes (comme `ls`, `pwd`, etc...) sera envoyé dans le fichier `/tmp/toto`. Evidemment, c'est peu intéressant de travailler de cette manière, par contre, on peut vouloir rediriger tous les messages d'erreur des commandes d'un programme vers un fichier particulier : dans ce cas, il suffit de placer en début de programme une commande du type :

```
exec 2>liste_erreurs
```

I.6.8. Il existe d'autres types de combinaison, mais qui sont bien moins usités.

Par exemple, il est possible de faire une redirection en mode entrée-sortie (`digit<>name`), mais le mode sortie est en fait un mode ajout, ce qui n'est pas vraiment intéressant. De toute façon, la quasi-totalité des commandes UNIX n'acceptent pas d'avoir des descripteurs d'entrée et de sortie qui pointent sur un même fichier : on sera presque toujours obligé d'utiliser un fichier intermédiaire pour simuler le mode "entrée-sortie" sur un fichier ouvert.

Un bon usage des redirections ci-dessus permet déjà beaucoup de manipulations d'entrées-sorties.

Exemple 1 : plus de trois descripteurs pour une commande

```
$ more essai1
while read -u4 ligne
do
      print -u6 $ligne
done
$ essai1 4<toto1.c
essai1[3]: print: bad file unit number
essai1[3]: print: bad file unit number
$ essai1 4<toto1.c 6>&1
ceci est la 1ere ligne du fichier toto1.c
ceci est la 2eme 1igne du fichier toto1.c
$
```

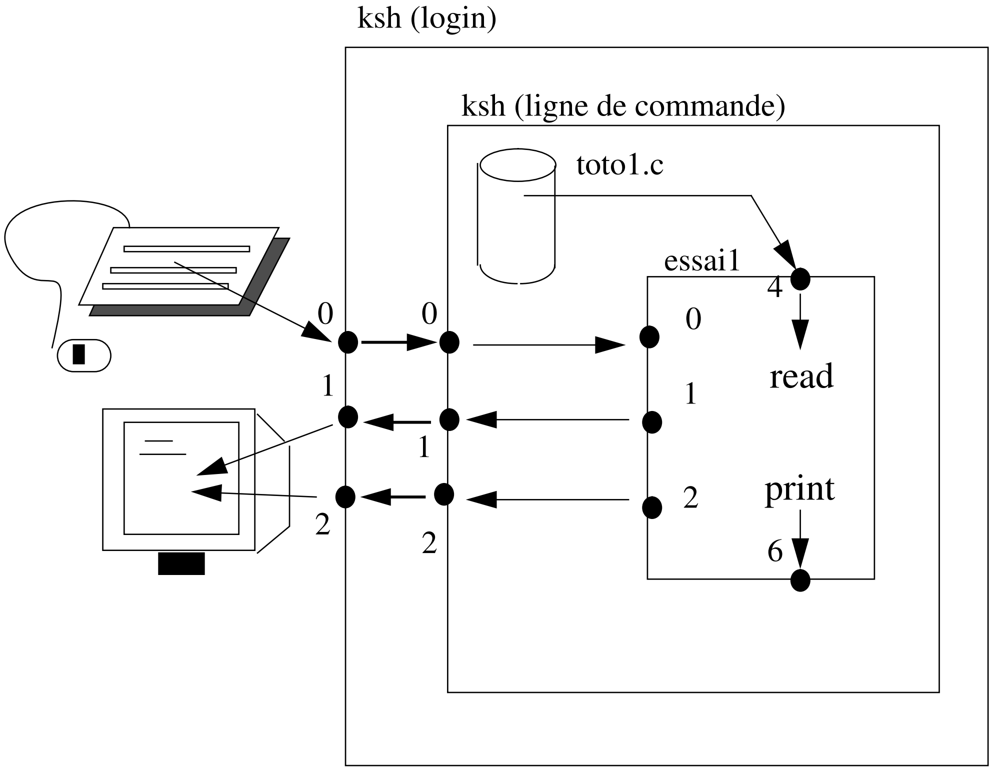
Pour cet exemple, on a écrit un petit programme shell (dont le fichier se nomme `essai1`), qui lit des lignes sur le descripteur 4 et les recopie sur le descripteur 6.

La première commande lancée (`essai1 4<toto1.c`) provoque une erreur ; notez que le numéro indiqué dans le message d'erreur est lié à la ligne où s'est produite l'erreur, et n'a rien à voir avec le descripteur 4[1].

1. voir le chapitre sur les messages d'erreur. page 205

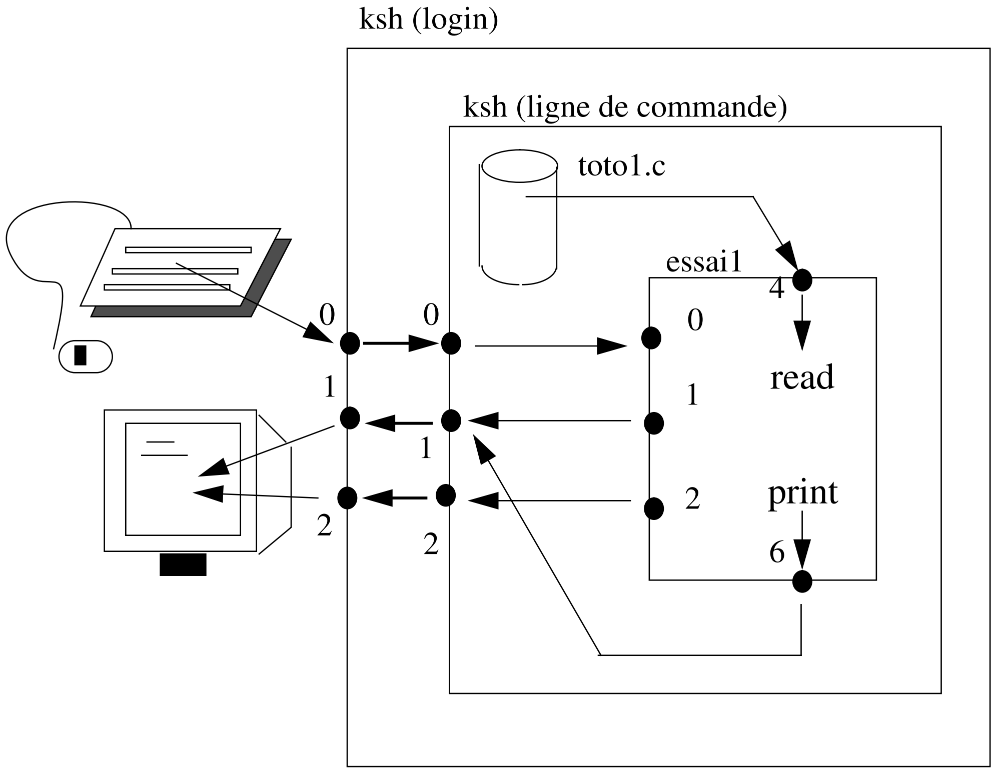
I.6.8.

On peut suivre l'état des redirections sur le schéma suivant :



Le script `essai1` est relié lors de son lancement au shell de la ligne de commande par les entrée-sorties standard, puisque aucune redirection sur un des descripteurs 0, 1 ou 2 n'a été spécifiée ; Cependant, essai1, de par sa construction, lit ses données sur le descripteur 4, vers lequel la redirection `4<toto1.c` fait pointer un fichier en lecture, et écrit sur le descripteur 6, qui n'est associé à aucun fichier physique : c'est l'erreur détectée.

I.6.8. Voici maintenant le schéma correspondant à la deuxième commande, où les redirections en entrée et en sortie sont spécifiées :



Le programme essai1 peut maintenant écrire sur le descripteur 6 qui est relié par la redirection `6>&1` au même fichier physique que le descripteur 1, c'est à dire la console.

Dans le cas ci-dessus, on voit que le shell a associé, comme toujours, les trois descripteurs standards 0, 1 et 2 au programme `essai1` (en plus des descripteurs 4 et 6), sans savoir s'ils seraient utilisés ou non. Ici, rien n'empêcherait de rajouter dans le programme `essai1` des commandes `echo` par exemple, qui écriraient sur le descripteur 1 (donc vers la console) : les cinq descripteurs sont disponibles.

<u>Exemple 2</u> : les redirections avec une macro-commande.

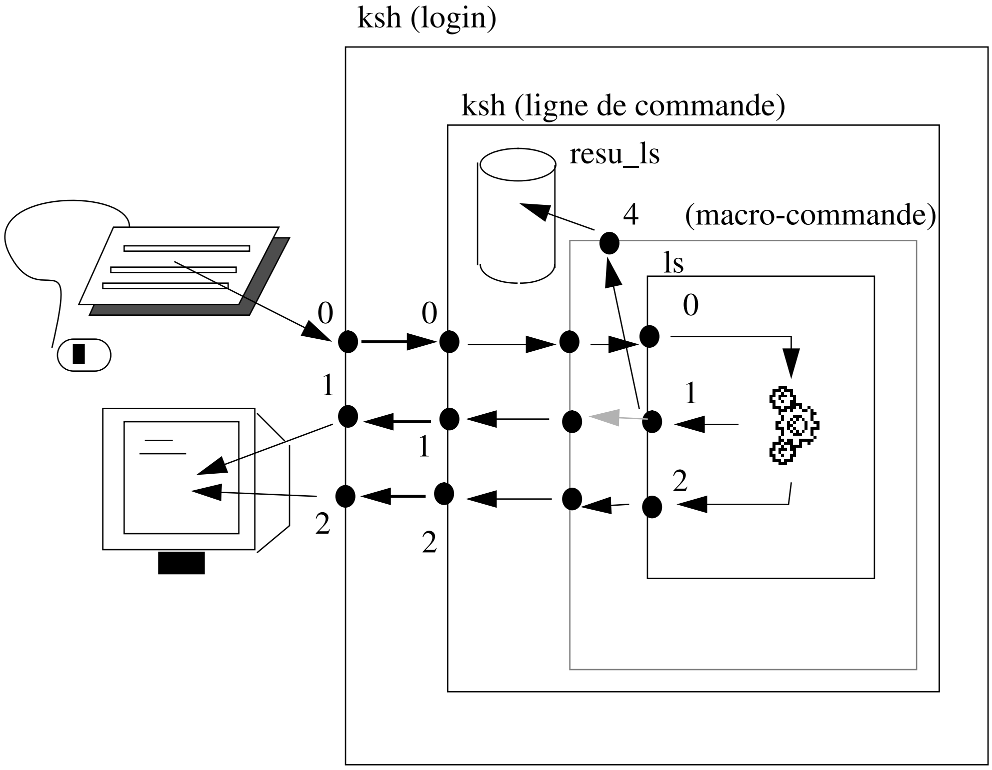
voir

voir

```
$ pwd
/users/logis/doc/manuell_shell/exemples
$ ( cd /usr/pub ; ls >&4 ) 4>resu_ls
$ pwd
/users/logis/doc/manuell_shell/exemples
$ more resu_ls
ascii eqnchar
$
```

La macro-commande ci-dessus permet dans le shell courant de créer un environnement séparé dans lequel sont exécutées les commandes `cd /usr/pub` et `ls`, cette commande envoyant son résultat sur le descripteur 4. Le shell redirige le descripteur 4 de la macro-commande vers le fichier physique resu_ls. I.6.8.

Voici le schéma correspondant :



On s'intéresse ici uniquement aux redirections de la commande `ls`, la commande `cd` ayant conservé les entrées-sorties standard. On constate que la macro-commande se comporte au vu des redirections exactement de la même manière qu'un script et qu'il est possible de multiplier le nombre de descripteurs.

En conclusion, on peut mettre sur une ligne autant de redirections que la commande concernée manipule de descripteurs, et vice-versa. Les commandes standards d'UNIX en utilisent généralement trois comme on l'a déjà dit ; mais ces exemples montrent qu'il est possible d'en créer avec plus de trois.

La limite de 20 [1] qui a été indiquée précédemment est le nombre maximum de fichiers ouverts qu'un processus peut posséder ; c'est une valeur paramétrable par l'administrateur de la station.

1. voir la "manipulation des fichiers" page 21

I.6.9.

I.6.9. Etape 9 : La localisation de la commande

A ce stade, on peut imaginer la ligne initiale une fois que toutes les transformations correspondant aux étapes 1 à 8 qui précèdent ont été réalisées. Le premier champ de la ligne est alors normalement la commande.

1.6.9.1. Les alias

Un alias est une correspondance entre deux chaînes de caractères. On peut renommer ou créer de nouvelles commandes en utilisant des alias.

La première chaîne est quelconque (mais sans séparateur) ; la deuxième est la chaîne (qui peut contenir des séparateurs) avec laquelle on remplace la première si celle ci est trouvée dans la ligne de commande.

Pour déclarer un alias, on exécute la commande :

`alias` *`ancien_terme=chaine_de_remplacement`*

Chaque fois que le premier champ d'une ligne est égal à *`ancien_terme`*, il est remplacé de manière littérale par la chaîne *`chaine_de_remplacement`*. La transformation n'est faite qu'une seule fois (pas de manière récursive). De plus comme la transformation est faite de manière littérale, il n'est pas possible de spécifier dans un alias des noms de variables ou des paramètres de position ($1, $2,...).

L'alias une fois déclaré est valable dans l'environnement courant, et dans les macro-commandes de type () si on spécifie l'option `-x` au moment de la déclaration. Si on désire qu'un alias soit valable dans tous les programmes et les interpréteurs shell démarrés, il faut placer sa déclaration dans le fichier de configuration du shell[1].

Pour consulter la liste des alias valables dans le shell courant, taper la commande :

`alias`

et pour supprimer un alias existant dans le shell courant :

`unalias` *`ancien_terme`*

Si un alias a le même nom qu'une commande interne au shell ou non, il la remplace.

1.6.9.2. Les fonctions

Une fonction est une macro-commande nommée, que l'on peut appeler avec des paramètres. Une fois déclarée, une fonction devient en quelque sorte une nouvelle commande du shell, qu'il est possible de manipuler comme les autres.

Pour déclarer une fonction, on procède de la manière suivante :

1. `.kshrc` : voir le paragraphe qui se rapporte à ce sujet page 163

I.6.9.

nom_de_la_fonction()

```
        macro-commande
```

La macro-commande[1] peut être de type () si on désire que la fonction s’exécute dans un environnement protégé, ou {} pour qu’elle s’exécute dans l’environnement courant, ce qui est en pratique le plus intéressant.

Dans les deux cas, les paramètres de position $1, $2, $... sont utilisables dans une fonction et correspondent aux différents paramètres d’appel.

Les problèmes de visibilité et de déclaration dans le shell sont identiques à ceux indiqués pour les alias.

Pour lister les fonctions déclarées dans le shell courant, vous pouvez taper la commande :

```
typeset -f
```

Les alias sont évalués avant les fonctions, ce qui veut dire que le résultat d’un alias peut être un nom de fonction déclaré dans le shell courant.

L’exemple suivant est une combinaison d’alias avec des fonctions, et permet de modifier le fonctionnement de la commande interne `cd` de manière à ce que le répertoire courant et le nom de la station sur laquelle vous travaillez s’affichent sur le bandeau qui surmonte les fenêtres de type `hpterm` (valables sur les stations hp).

Il faut placer ces lignes dans votre fichier .kshrc pour que chacun de vos shells intéractifs démarré dans une fenêtre positionne cette combinaison. Si vous travaillez aussi avec d’autres types

1. Voir le paragraphe sur les macro-commandes: page 46

I.6.9. de fenêtres ou de terminaux, il vous faudra rajouter un test, ou lancer le fichier de configuration “à la main” uniquement dans les fenêtres `hpterm`.

voir

voir

```
#
# affichage sur une fenetre hpterm du repertoire
# et de la machine courante
#
export HOSTNAME=$(hostname)


xname()
{
 _title="$*"
 echo "\033&f0k${#_title}D${_title}\c";
 echo "\033&f-1k${#_title}D${_title}\c;
}

cdx()
{
 unalias cd
 cd $1;
 alias cd=cdx
 xname $HOSTNAME [pwd=$PWD]
}

alias cd=cdx
set +u
cd
```

Explications :

On déclare ici deux fonctions `xname` et `cdx`, et un alias qui remplace la commande interne `cd`.

`xname()` : cette fonction affiche sur le bandeau d'une fenêtre hpterm les arguments qui lui ont été passés à l'appel.

Les deux commandes `echo` envoient sur la sortie standard, c'est à dire la console hpterm des suites de caractères (<escape>&f0k...) qui indiquent une action particulière à effectuer, et qui ne sont pas affichés. Dans le premier cas, cela consiste à modifier le bandeau de la fenêtre si elle est ouverte, dans le deuxième son nom si elle est iconifiée.

cdx() : Cette fonction exécute la commande `cd` standard, puis appelle la fonction `xname` pour mettre à jour le bandeau de la fenêtre. Pour éviter une boucle infinie, il faut supprimer l'alias pour exécuter réellement la commande `cd`, puis le remettre pour reprendre la configuration initiale. (En effet, `cd` est aliasé en `cdx`, elle même fonction faisant appel à la commande `cd`...)

A ce niveau, on a déclaré deux fonctions ; pour mettre en pratique notre nouvelle fonctionnalité de la commande cd, on rajoute un alias qui remplace la commande tapée `cd` en `cdx`. A partir de ce moment, lorsque on tape `cd` *nom_de_repertoire*, le répertoire courant est changé et son nom apparait sur le bandeau de la fenêtre.

Si vous préférez avoir une commande `cdx` dont le nom reste distinct de la commande interne `cd`, il suffit de supprimer partout les `alias` et `unalias cd`. I.6.9.

La commande `set +u` rajoutée à la fin supprime les messages d'erreur lorsque on essaie d'utiliser une variable non positionnée : c'est le cas ici lorsqu'on tape la commande `cd`, pour revenir dans le répertoire principal ; la commande `cd` n'ayant pas d'argument, la fonction cdx n'en a pas non plus, et la variable `$1` n'est pas positionnée, ce qui provoque en tant normal un message d'erreur du shell.

Une fois déclarée, la fonction `xname`() par exemple, peut être utilisée seule, comme une commande normale, ainsi que n'importe quelle fonction ou alias existant dans le shell courant. Ainsi, en tapant :

```
xname essais
```

le bandeau de la fenêtre s'intitule `essais` (ainsi que l'icône une fois la fenêtre refermée).

Attention :

Si une fonction a le même nom qu'une commande élémentaire UNIX, elle la remplace ; ce n'est pas le cas si elle a le même nom qu'un alias ou une commande interne du shell qui gardent leur préséance.

1.6.9.3. Le path

La variable `$PATH` contient la liste des répertoires dans lesquels le shell recherche le fichier exécutable correspondant au nom d'une commande élémentaire.

Une fois toutes les transformations (vues ci-dessus) effectuées, le premier champ de la commande doit être :

une commande interne :

auquel cas elle est exécutée avec comme arguments les champs qui la suivent ;

autre chose :

auquel cas le premier champ est considéré comme un nom de fichier exécutable.

Si le premier champ comporte un caractère /, le shell va chercher le fichier dans le répertoire indiqué ;

Si le premier champ ne comporte pas de caractère /, le shell va chercher dans un des répertoires qui apparaissent dans la variable `$PATH` le fichier exécutable.

Le contenu de la variable $PATH a la forme suivante :

répertoire1 **:** *répertoire*2 **:** *répertoire*3 **:** etc...

Cette variable est toujours positionnée par défaut avec une liste minimale ; mais si vous faites appel à des commandes particulières, ou que vous avez créées vous-même, il suffit de rajouter les répertoires correspondant de la façon suivante :

PATH=$PATH **:** *répertoire1* **:** *répertoire*2 **:** *répertoire*3 **:** etc...

I.6.9. La recherche est effectuée dans l'ordre d'apparition ; ce qui veut dire que si deux fichiers exécutables portent le même nom, mais sont placés dans deux répertoires différents, celui qui sera lancé est celui qui apparait dans le répertoire cité en premier dans la liste du PATH.

Parmi les répertoires, vous pouvez mettre le répertoire courant qui est symbolisé par le caractère point ; il existe deux philosophies lorsqu'on veut utiliser des commandes dont l'exécutable se trouve dans le répertoire courant :

PATH=$PATH : .

dans ce cas, une commande dont le fichier exécutable est placé dans le répertoire courant est exécutée uniquement si un fichier (une commande) de même nom n'existe pas dans les répertoires placés avant dans la liste ;

PATH=. : $PATH

dans ce cas, c'est le contraire : les commandes dont l'exécutable se trouve dans le répertoire courant sont exécutées en priorité.

Cette déclaration peut provoquer des confusions : une commande n'a plus la même signification suivant le répertoire dans lequel on la tape.

Remarque : seuls sont considérés dans les répertoires donnés dans $PATH les fichiers de type exécutable, c'est à dire dont les droits apparaissent en `x` (lors d'une commande `ls -l`) pour la classe de l'utilisateur.

I.7. Le contrôle des processus

I.7.1.

I.7.1. La commande ps

La commande `ps` permet de lister les processus actifs :

voir

voir

```
$ ps -fu logis
 UID   PID   PPID  C STIME    TTY   TIME COMMAND
 logis 25723 24887 4 11:44:21 ttyu0 0:00 ps -fu logis
 logis 24887 24886 1 07:53:50 ttyu0 0:02 -ksh
```

Utilisé de cette manière, la commande `ps` affiche la liste des processus appartenant à l'utilisateur `logis :`

UID : c'est le propriétaire du processus. Toute action sur ce processus ne peut être faite que par lui ou par le super-utilisateur.

PID : c'est le numéro du processus correspondant. Un processus est toujours rattaché à un numéro unique sur une machine.

PPID : c'est le numéro du processus père (celui qui a lancé le processus *PID*) ; si le processus père meurt avant son fils, ce dernier est alors rattaché au processus 1. (PPID vaut 1) On peut ainsi suivre toute l'arborescence des processus en chaînant les valeurs PID et PPID.

STIME : date de démarrage du processus

TIME : temps d'exécution en minutes et secondes CPU depuis le démarrage du processus

COMMAND : intitulé de la commande correspondant au processus

TTY : le nom de la console à laquelle le processus est rattaché. Tous les processus lancés en intéractif sont rattachés à une console ou un terminal : si Xwindows fonctionne, le terminal correspond alors à une fenêtre, sinon c'est la console elle-même. Sauf manipulation particulière, un processus est rattaché à la même console que son père.

En contrepartie, les processus lancés en mode "batch" (par des commandes comme `at` ou `batch`) et les processus serveurs (les "daemons") ne sont rattachés à aucun terminal[1].

Le fait qu'un processus soit rattaché à un terminal indique qu'il est susceptible de réagir à certains événements liés aux terminaux, c'est à dire, certaines frappes de caractères de contrôle, les déconnexions, etc...

Ces indications sont suffisantes dans la très large majorité des cas pour gérer ses processus.

1. le nom de la console est alors remplacé par un **?** ou un **-**

I.7.2. La commande `ps` est intéressante à utiliser de deux manières :

`ps -fu` *`nom_utilisateur`*

pour connaître les processus attachés à un utilisateur particulier ;

`ps -ef`

pour connaître tous les processus tournant sur le système.

Sachez qu'un processus tel le serveur X11 (intitulé : X : 0) qui est très gourmand et tourne souvent toute la journée, utilise par jour une dizaine de minutes de temps CPU, que certains processus comme les compilations ada atteignent quelques minutes d'utilisation, mais que la grande majorité des processus ne dépassent pas quelques secondes dans leur "vie".

I.7.2. Le contrôle des processus par le clavier

Une fenêtre X11 est en fait une émulation d'une console écran-clavier. Donc, la plupart des fonctionnalités d'une console sont supportés : on peut avoir le nom de la console en tapant la commande `tty` (l'écran nu sans utilisation d'X11 s'appelle `console`), et ses caractéristiques (type d'émulation, écho, ...) par la commande `stty -a`.

Une des caractéristiques d'une console est de posséder des séquences de caractères qui permettent d'intéragir par l'intermédiaire de signaux avec les processus qui lui sont rattachés : en général, la séquence est composée de la touche <control> et d'une autre touche. Pour associer une séquence de caractères avec un signal, on utilise la commande `stty` qui configure la console ou le pseudo-terminal si on travaille avec X11.

Voici les séquences couramment utilisées et leur fonctionnalité :

<control>C cette séquence est la plus connue : son effet consiste à envoyer un signal SIGINT d'interruption au processus en avant-plan (foreground), autrement dit celui qui est en cours d'exécution. Dans la majorité des cas, l'envoi du signal SIGINT provoque la mort du processus en cours et le shell rend la main en affichant le prompt.

Pour positionner la séquence :

```
stty intr ^c
```

<control>Z cette séquence envoie un signal SIGTSTP au processus en avant-plan ; ce signal suspend l'exécution du processus dans l'état en cours et rend la main au shell.

Grâce aux commandes fg et bg, il est possible de reprendre l'exécution du processus, soit en avant-plan, soit en arrière plan.

Pour positionner la séquence :

```
stty susp ^z
```

I.7.3.

<control>S cette séquence suspend le défilement de l'affichage à l'écran, mais le processus continue de s'exécuter (sauf si le buffer d'affichage est plein).

Pour positionner :

```
stty stop ^s
```

<control>Q permet de reprendre le défilement de l'écran.

Pour positionner :

```
stty start ^q
```

<control>D spécifie au processus dont l'entrée standard est la console que la fin de fichier est atteinte. On tape cette séquence par exemple après avoir fourni à l'entrée standard d'une commande le contenu du buffer de la souris.

Pour positionner :

```
stty eof ^d
```

<erase> spécifie au shell la séquence qui correspond à l'effacement d'un caractère sur la ligne de commande.

Pour positionner :

```
stty erase <backspace>
```

ou `stty erase <delete>`

suivant le type de votre clavier.

Les séquences qui sont indiquées ici sont celles qui sont positionnées normalement lorsque vous vous connectez à une console ou à un pseudo-terminal (les commandes `stty` sont placées dans le fichier `.profile` de configuration de login), mais rien ne vous empêche de les modifier, ou de les compléter si votre configuration est incomplète.

I.7.3. La commande `jobs`

Grâce au shell intéractif, il vous est possible de lancer des commandes (c'est à dire des processus) en avant-plan (foreground), ou en arrière-plan (background) si vous rajoutez l'opérateur & .

Comme il ne peut y avoir pour un shell intéractif qu'un processus en avant-plan à la fois, on comprend qu'il est facile de le gérer, en particulier grâce aux caractères de contrôle vus ci-dessus.

Par contre, il est possible d'avoir simultanément plusieurs processus en arrière-plan, lancés à partir du même shell intéractif.

Dans ce cas, on contrôle ces processus en obtenant leur identifiant et leur état par la commande `jobs`, puis éventuellement en modifiant leur état par les commandes `fg`, `bg` ou `kill`.

I.7.4. Exemple :

voir

```
$ sleep 1000
<control>Z
[1] + Stopped sleep 1000
$ sleep 2000 &
[2] 2012
$ jobs
[2] + Running sleep 2000 &
[1] - Stopped sleep 1000
$
```

Voici la description de la manipulation réalisée :

1/ un processus est lancé en avant-plan (`sleep 1000`)

2/ on tape la séquence de contrôle `<control>Z`[1], le processus est stoppé, et le shell donne un message indiquant :

`[1]` : le descripteur du processus ; chaque processus géré ainsi par le shell possède un numéro qui permet de le désigner par la suite.

`Stopped` : l'état du processus. On peut avoir “stopped” qui indique que le processus est en sommeil, “running” qui indique que le processus s'exécute, “done” qui indique que le processus vient de se terminer, et “terminated” qu'on vient de le tuer. Dans les autres cas, le type d'état est lié au type du signal que le processus vient de recevoir.

`sleep 1000` : l'intitulé de la commande correspondant au processus.

3/ un deuxième processus est lancé (`sleep 2000`), mais celui-ci directement en arrière-plan. Le shell indique :

`[2]` : le descripteur du nouveau processus

`2012` : l'identificateur de processus, tel qu'il apparait si vous listez les processus avec la commande `ps`. Attention, pas de confusion : le descripteur est un numéro propre au shell, alors que l'identificateur est le numéro qui définit de manière unique le processus dans le système.

4/ on tape la commande `jobs` qui liste les processus en arrière-plan contrôlés par le shell.

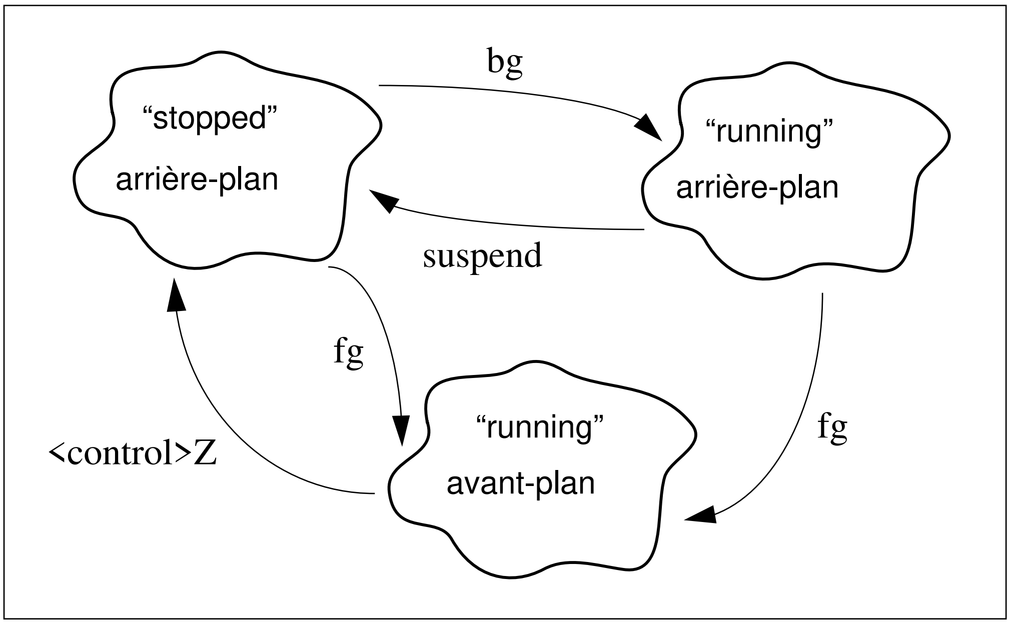
I.7.4. Les commandes **`fg`** et **`bg`**

La commande `fg %`*`id`* permet de faire passer un job d'identificateur *`id`* de l'état “stopped” ou “running” en arrière-plan vers l'état “running” de l'avant-plan.

1. pour que la frappe du <control>Z ait un effet, il faut que la console ait été configurée pour y réagir ; voir le paragraphe qui précède.

La commande `bg %id` permet de faire passer un job d'identificateur *id* de l'état "stopped" en arrière-plan vers l'état "running" de l'arrière-plan. I.7.4.

Schématisation :



La commande suspend n'existe pas en standard, mais on peut la définir avec un alias :

```
alias suspend='kill -STOP'
```

Nous allons voir l'effet de la commande `kill` au paragraphe suivant.

Exemple :

voir

```
$ sleep 1000
<control>Z
[1] + Stopped sleep 1000
$ bg %1
[1] sleep 1000&
$ suspend %1
[1] + Stopped (signal) sleep 1000
$ bg %1
[1] sleep 1000&
$ fg %1
sleep 1000
```

Voici les différentes étapes :

1/ création du processus `sleep 1000` en avant-plan

2/ frappe de la séquence *<control>Z* qui fait passer le processus vers l'arrière-plan en mode "stopped"

I.7.5.

3/ la commande `bg` fait passer le processus de l'état “stopped” vers l'état “running”, toujours en arrière-plan.

4/ la commande `suspend` refait passer le processus en mode “stopped”

5/ `bg` repasse le processus en mode “running” de l'arrière-plan

6/ la commande `fg` fait passer le processus en mode “running” de l'arrière-plan vers l'avant-plan

Remarque :

La commande `jobs` liste les processus, précédés d'un caractère **+** ou **-**. Une commande `bg` ou `fg` sans paramètre s'appliquera au seul processus indiqué avec un **+** (en général, c'est le dernier créé ou manipulé).

I.7.5. La commande `kill`

Contrairement à son nom, cette commande n'est pas destinée réellement à tuer un processus, mais plutôt à lui envoyer un signal.

Il existe un certain nombre de signaux que l'ont peut émettre vers un processus :

voir

voir

$ kill -l

1) HUP	16) USR1
2) INT	17) USR2
3) QUIT	18) CHLD
4) ILL	19) PWR
5) TRAP	20) VTALR
6) IOT	21) PROF
7) EMT	22) IO
8) FPE	23) WINDO
9) KILL	24) STOP
10) BUS	25) TSTP
11) SEGV	26) CONT
12) SYS	27) TTIN
13) PIPE	28) TTOU
14) ALRM	29) URG
15) TERM	30) LOST

On peut rajouter deux autres signaux qui sont ignorés par le shell en temps normal :

0) EXIT qui est déclenché en fin de programme shell

35) DEBUG qui est déclenché après chaque commande

Pour envoyer un signal à un processus, on utilise la commande suivante :

kill -*`signal No_process`*

I.7.5.

`signal` est soit le numéro, soit la désignation du signal telle quelle apparait ci-dessus. Si `-signal` est omis, c'est -TERM qui est pris par défaut.

`No_process` vaut :

- `%descripteur`, avec `descripteur` égal à un numéro de descripteur donné par le shell,

  ou bien directement :

- `identificateur`, avec `identificateur` égal au numéro d'identification du processus, tel qu'il apparait avec la commande `ps`.

Le processus qui reçoit un signal peut réagir de trois manières différentes :

a/ le processus a été configuré lors de son démarrage pour ignorer le signal donné : aucun effet

b/ le processus a été configuré lors de son démarrage pour attraper le signal donné : à la réception du signal, le processus exécute un déroutement vers la fonction associée.

c/ rien n'a été prévu : le processus meurt. Dans ce cas, le processus peut être tué par n'importe quel signal.

Un cas particulier existe avec le signal 9 (KILL) : il est impossible de l'ignorer ou de l'attraper : on est donc sûr de tuer un processus en lui envoyant un signal KILL.

Un processus peut recevoir un signal d'un autre processus, (c'est ce qui se passe par exemple quand vous tapez la commande `kill`), de la console (si vous tapez une séquence de contrôle) ou du hardware ("Bus error" par exemple)

Exemple :

voir IBM

```
$ sleep 1000 &
[1] 3800
$ sleep 2000 &
[2] 3801
$ jobs
[2] + Running   sleep 2000 &
[1] - Running   sleep 1000 &
$ kill -9 %1
[1] - Killed    sleep 1000 &
$ kill -INT 3801
[2] + Interrupt sleep 2000 &
$ jobs
$
```

On a utilisé la première fois le signal 9 (KILL), et la deuxième le signal 2 (INT), qui correspond au signal émis par la console lorsque vous tapez la séquence <*control*>C : c'est une bonne façon pour essayer de tuer un processus "proprement" : sur réception d'un signal 9, un processus meurt immédiatement, alors que si le signal est INT par exemple, une commande bien écrite peut toujours se dérouter pour supprimer ses fichiers ou objets temporaires avant de s'arrêter.

I.7.5. Il est possible d'attraper ou d'ignorer des signaux depuis un programme shell ; un exemple d'utilisation est donné plus loin[1] grâce à la commande shell `trap`.

Vous pouvez trouver une signification succincte des signaux utilisés sur votre machine en explorant le fichier `/usr/include/sys/signal.h`.

1. voir chapitre page 86

I.7.5.

II. Les Expressions

II.1.1.

II.1. Les expressions du KornShell

II.1.1. L'expression arithmétique

Les indices de tableau, les variables de type entier (voir `typeset`) peuvent être affectées avec une expression arithmétique.

L'expression arithmétique est une combinaison de signes arithmétiques + - * / , de nom de variables de type entier et de parenthèses :

Exemples :

val*10+1

'(val * val) * 2'

On rajoute les simple-quotes lorsqu'on utilise des parenthèses, ou que l'on a placé des espaces entre les termes de l'expression. Il n'est pas nécessaire de mettre devant les noms de variable le symbole $.

Utilisation : (on note <exp_arith> une expression arithmétique quelconque)

- affectation d'une *variable* de type entier ou indexation d'un *tableau*

  variable=<exp_arith>

  `echo` ${*tableau*[<exp_arith>]}

- affectation d'une *variable* de type non-entier ou obtention du résultat

  variable=$((<exp_arith>))

  `echo` $((<exp_arith>))

II.1.2. L'expression générique

On a vu, en étudiant la transformation d'une ligne de commande[1], la façon dont les caractères spéciaux *,[],?, etc..., pouvaient correspondre à des parties de nom de fichiers présents dans un répertoire donné.

Une chaîne de caractères qui comprend ce type de caractères spéciaux est appelée *expression générique*, car elle désigne en fait une classe de noms.

Lorsque une expression générique est employée à la place d'un nom de fichier sur la ligne de commande, le shell instancie la classe correspondante avec tous les fichiers dont le nom s'adapte à la classe définie.

Mais on peut aussi utiliser des expressions génériques dans des tests lorsque vous utilisez les commandes *`if`*, *`case`* ou *`while`*. Dans ce cas, la partie droite du test classique :

```
if [ valeur1 = valeur2 ]
...
```

1. Voir page 43

II.1.3. peut être une expression générique, à condition d'utiliser l'opérateur [[ ]] au lieu des [ ] classiques.

Exemple[1] :

```
$ [ toto = toto ] && print ok
ok
$ [ toto = t* ] && print ok
$ [[ toto = t* ]] && print ok
ok
$ [[ toto = a* ]] && print ok
$
```

L'utilisation avec if , case ou while se fait de la même façon : vous aurez des exemples dans les paragraphes dédiés à ces commandes : voir page 82

Voici un rappel de la combinaison des expressions:

?(*expression*)

?(*expression1*|*expression2*|...)

remplace zéro ou une occurrence des expressions fournies.

*(*expression*)

*(*expression1*|*expression2*|...)

remplace zéro ou plusieurs occurrences des expressions.

+(*expression*)

+(*expression1*|*expression2*|...)

remplace une ou plusieurs occurrences des expressions.

@(*expression*)

@(*expression1*|*expression2*|...)

remplace une occurrence exactement d'une des expressions fournies.

!(*expression*)

!(*expression1*|*expression2*|...)

est remplacé par toutes les occurrences possibles sauf celles correspondant aux expressions fournies.

II.1.3. L'expression conditionnelle

Une expression conditionnelle est principalement utilisée avec la commande `if` pour les structures conditionnelles et la commande `while` pour les boucles. Son évaluation donne un résultat vrai (= 0) ou faux (<> 0).

1. Voir page 74 le détail de l'opérateur && (*et* logique entre deux commandes)

II.1.3.

Pour pouvoir donner des exemples pratiques, on va tout d'abord désigner à partir d'ici une expression conditionnelle par <exp_cond>, et l'opérateur **[]** :

[<exp_cond>**]** l'opérateur [ ] est une commande appliquée sur l'expression <exp_cond> et dont le seul effet est de renvoyer un code de retour égal à zéro si l'expression est vraie ou différent de zéro si l'expression est fausse.

De plus, nous allons aussi utiliser les séquences && et || dont voici un rappel du fonctionnement :

cmd1 **&&** cmd2

la commande cmd2 est lancée si le code de retour de la commande cmd1 est zéro (commande réussie)

cmd1 **||** cmd2

la commande cmd2 est lancée uniquement si cmd1 a échouée (code de retour non nul)

Exemple :

```
$ [ 0 = 0 ] || print ok
$ [ 0 = 1 ] || print ok
ok
$ [ 1 = 1 ] && print ok
ok
$ [ 1 = 2 ] && print ok
$
```

Ici, la première commande correspond à l'évaluation de l'expression conditionnelle, grâce à l'opérateur [ ] , et la seconde est la commande `print ok`.

Une expression conditionnelle <exp_cond> peut prendre une des formes suivantes :

- un test sur un fichier :

`-a` *fichier* vrai si le fichier *fichier* existe

`-d` *fichier* vrai si *fichier* est un répertoire

...

Il y en a comme ça une bonne vingtaine[1].

```
$ ls -l travail
-rw-r--r-- 1 logis users 143 Jun 22 10:41 travail
$ [ -a travail ] && print ok
ok
$ [ ! -a absent ] && print inconnu
inconnu
```

1. on peut avoir la liste complète dans la section 1 du manuel de référence UNIX, à la rubrique “ksh”, ou avec le manuel en ligne, en tapant: `man ksh`

II.1.3.

- un test sur une variable ou chaîne de caractères :

  `-z` *chaîne* vrai si la longueur de *chaîne* est nulle

  `-n` *chaîne* vrai si la longueur de *chaîne* est non nulle

```
$ chaine="coucou"
$ [ -n "$chaine" ] && print non vide
non vide
$ [ -z "" ] && print vide
vide
$
```

- un test entre deux fichiers :

  fic1 `-nt` *fic2* vrai si *fic1* a été modifié plus récemment que *fic2*

  ...

```
$ ls -l travail tutu
-rw-r--r-- 1 logis users 143 Jun 22 10:41 travail
-rw-r--r-- 1 logis users  78 Apr 15 15:34 tutu
$ [ travail -nt tutu ] && print travail est plus jeune
travail est plus jeune
$
```

- un test entre deux variables ou chaînes de caractères

  chaîne = <expression_générique>

```
$ reply=oui
$ [ "$reply" = oui ] && print oui
oui
$ [ "$reply" = [Oo]* ] && print oui
$ [[ "$reply" = [Oo]* ]] && print oui
oui
$ reply=OUI
$ [[ "$reply" = [Oo]* ]] && print oui
oui
$
```

Remarques : l'utilisation d'une expression générique nécessite d'utiliser des [[ ]] à la place des [ ] simples.

D'autre part, la présence du caractère = dans le test force à <chaîne> le type des arguments fournis; les tests sur des valeurs numériques se font avec les opérateurs `-ne`, `-ge`, etc.. au lieu du =.

II.1.3.

• un test entre deux expressions arithmétiques.

```
$ [ 3*2+1 -eq 1+2*3 ] && print ok
ok
$ [ 3 *2+1 -eq 1+2*3 ] && print ok
ksh: *2+1: unknown test operator
$ [ '3 * (2 + 1)' -gt 1+2*3 ] && print ok
ok
$
```

Voici les différents tests arithmétiques :

-eq	égalité
-ne	différent
-gt	supérieur à
-lt	inférieur à
-ge	supérieur ou égal à
-le	inférieur ou égal à

On peut combiner plusieurs expressions conditionnelles pour en fabriquer une plus complexe (pour la suite, on note <exp_cond> une expression vue ci-dessus) :

<exp_cond1> && <exp_cond2>

cette expression est vraie si les deux sont vraies

<exp_cond1> || <exp_cond2>

cette expression est vraie si l'une des deux est vraie

! <exp_cond>

cette expression est vrai si <exp_cond> est fausse

Exemples :

```
$ echo $reply
OUI
$ [ ! 3 -eq 2 -a "$reply" = OUI ] && print ok
ok
$ [ ! 3 -eq 2 && "$reply" = OUI ] && print ok
ksh: test: ] missing
$ [ ! 3 -eq 2 \&\& "$reply" = OUI ] && print ok
ok
$ [ ! 3 -eq 2 '&&' "$reply" = OUI ] && print ok
ok
$ [[ ! 3 -eq 2 && "$reply" = OUI ]] && print ok
ok
$
```

II.1.3. Vous remarquerez qu'il peut y avoir sur la même ligne, comme ici, des opérateurs **&&** liés à des commandes, et des opérateurs **&&** liés à des expressions. Pour ne pas que le shell confonde les deux, il faut :

- protéger par des ****, des `""`, ou des `''` les opérateurs && destinés aux expressions[1], ou alors les faire figurer dans des **[[ ]]** plutôt que des **[]**.
- ne pas protéger les opérateurs **&&** destinés aux commandes.

La remarque vaut aussi pour l'opérateur **||**.

Pour plus de deux expressions, on groupe avec des parenthèses () :

```
$ [ 3 -eq 2*3 '&&' \( "1" = "2" '||' 3 -eq 3 ')' ] && print ok
$ [[ 3 -eq 3 && ( "1" = "2" || 3 -eq 3 ) ]] && print ok
ok
$
```

Le problème de protection des ( ) est le même que pour les && ou les `||`.

1. lire dans le paragraphe sur l'interprétation des lignes de commande la façon dont les caractères spéciaux du shell conservent (ou perdent) leur signification particulière : voir page 35

II.2. Les expressions régulières II.2.1.

Les expressions régulières sont pour les commandes Unix l'équivalent des expressions génériques pour le shell : ce sont des chaînes de caractères (éventuellement spéciaux) qui représentent des classes de chaînes de caractères.

A une expression régulière donnée, on peut associer un certain nombre de chaînes de caractères, qui ont toutes des caractéristiques communes : celles de l'expression régulière.

Les commandes qui utilisent les expressions régulières (appelées aussi *pattern*) sont :

- `vi` , `ex`
- `ed`, `sed`, `expr`
- `grep`, (ou `egrep`)
- `awk`
- etc...

Elles permettent pour ces commandes de repérer dans du texte des suites de caractères ayant des caractéristiques particulières.

Attention[1] **:**

- Bien que se rapprochant beaucoup de la syntaxe des expressions génériques reconnues par le shell, les expressions régulières ont une syntaxe spécifique, et qui n'est pas compatible.

voir

- Il existe des expressions régulières de base, et des expressions régulières *étendues*, qui sont un sur-ensemble des premières (!!) .

voir

Les premières sont acceptées par toutes les commandes indiquées ci-dessus, et les secondes uniquement par le `awk`, et le `grep` avec l'option `-E`, (ou le `egrep`).

La référence peut être trouvée pour les deux types d'expressions dans le manuel(5) UNIX, au terme **`regexp`**, ou dans le manuel en ligne (taper : `man regexp`) ;

Les explications générales qui suivent en sont tirées.

II.2.1. Expressions régulières de base (ER)

caractères ordinaires

Un caractère ordinaire est n'importe lequel des caractères alphanumériques, excepté les caractères spéciaux qui suivent ci-dessous, et le caractère <newline>.

Un caractère ordinaire correspond à sa propre valeur.

ER	correspond à
a	a

1. tout ce qui est complexe est encore beaucoup trop simple.

II.2.1.

%	%
abc	abc
\b	b

un caractère ordinaire précédé d'un **** conserve sa signification, excepté **(**, **)**, **{**, **}**, et les chiffres de **1** à **9** (voir ci-dessous, ainsi qu'à la commande `sed`)

caractères spéciaux

les caractères spéciaux ont une signification particulière, (qu'ils perdent si on les fait précéder d'un ****) :

**** => annule la signification particulière du caractère suivant (sauf pour **(**, **)**, **{**, **}**, et les chiffres de **1** à **9)**

ER	correspond à
\.	.
\\\\	\\
\$	$

. => peut prendre n'importe quelle valeur de caractère

ER	correspond à
.	*a* ou *b* ou *c* ... ou *A* ou *B* ...ou *1* ou *2* ...
..	*aa* ou *ab* ou *Aa* ou *A1* ou ...

[ ] => peut prendre n'importe quelle valeur de caractère parmi ceux spécifiés entre les crochets

ER	correspond à
[abd9]	*a* ou *b* ou *d* ou *9*
[a-z]	*a* ou *b* ou *c* ... ou *z*
[a-z_1]	un caractère compris entre *a* et *z* comme ci-dessus ou alors un _ ou alors un *1*
[ab][12]	*a1* ou *a*2 ou *b1* ou *b2*
[^a-z1]	n'importe quel caractère sauf les caractères de *a* à *z* et sauf le *1*
z[ab]z	*zaz*, ou *zbz*

----

* => une séquence ER* correspond à une chaîne composée de 0 ou n répétitions de l'expression régulière ER

ER	correspond à

II.2.1.

a*	<rien> ou *a* ou *aa* ou *aaa* ou *aaaa* ...
[ab]*	une série (éventuellement nulle) de *a* et de *b* dans n'importe quel ordre
[ab][ab]*	idem précédent, mais la série comporte au moins un *a* ou un *b*
[^ab]*	une série de n'importe quel caractère sauf a et b, dans n'importe quel ordre, ou alors <rien>
z[a]*z	*zz* ou *zaz* ou *zaaz* ou *zaaaz* ou ...
.*	<rien> ou alors une suite d'un nombre quelconque de n'importe quel caractère.

----

^ => une séquence ^ER correspond à une chaîne *placée en début de ligne* et vérifiant l'expression ER

<u>ER</u>	<u>correspond à</u>
^a	une ligne commençant par un *a*
^toto	une ligne commençant par *toto*
^[\t]*--	une ligne commençant par des <espace> ou <tabulations> répétés 0 ou n fois, suivis par --

----

**$** => une séquence ER$ correspond à une chaîne *placée en fin de ligne* et vérifiant l'expression ER

<u>ER</u>	<u>correspond à</u>
a$	une ligne finissant par un *a*
toto$	une ligne finissant par *toto*
[\t]$	une ligne finissant par un <espace> ou une <tabulation>
^$	une ligne vide
^[\t]*$	une ligne vide ou alors ne contenant que des <espace> ou <tabulation>
^.*$	la totalité de la ligne, quelque soit son contenu.

\{ \} => une séquence ER\{m,n\} correspond à une chaîne vérifiant l'expression ER, répétée au minimum m fois et au maximum n.

voir

voir

=> ER\{m,\} : il doit y avoir au minimum m occurrences de ER

=> ER\{m\} : il doit y avoir exactement m occurrences de ER

<u>ER</u>	<u>correspond à</u>
za\{2,4\}z	*zaaz* ou *zaaaz* ou *zaaaaz*
[A-Z]\{3\}	une séquence de trois majuscules consécutives.

©jf pujol, 18 août 1993

II.2.2.

ATTENTION:

Tel qu'est fait le mécanisme de recherche dans les chaînes, il faut se méfier lorsqu'on écrit une expression régulière devant reconnaître un nombre exact d'occurrences:

Par exemple, l'expression `a\{2\}` est censée correspondre à une chaîne contenant exactement 2 lettres `a` consécutives; et pourtant, la chaîne `zaaaz` sera reconnue.

Cela est dû au fait que le mécanisme considère la chaîne `zaaaz` comme un `z`, suivi par exactement 2 lettres `a`, suivi par la séquence `az`: ce n'est pas tout à fait ce que l'on recherchait !

Pour que la recherche soit correcte, il faut inclure dans l'expression quelque chose de reconnaissable avant et après la séquence répétée: ici, on mettrait: `za\{3\}z`, qui correspond alors à exactement 3 lettres `a` encadrées par une lettre `z`; il ne peut plus y avoir d'ambiguïté.

II.2.2. Expressions régulières étendues (ERE)

Les expressions étendues reprennent les caractéristiques des expressions régulières de base (ER), plus les suivantes :

+ => une séquence ERE**+** correspond à une chaîne composée d'une ou n répétitions de l'expression régulière ERE

? => une séquence ERE? correspond à une chaîne composée de zéro ou une répétition de l'expression régulière ERE

( ) => une séquence (ERE*ERE*) compose une expression régulière plus complexe à partir de celles élémentaires

voir

ERE	correspond à
z*(ab)?z*	*zz* ou *zabz*
([abc]z)\{3\}	*azbzcz* ou *azczbz* ou *bzazc*z ou *bzczaz* ou *czazbz* ou *czbzaz*

| => une séquence (ERE|ERE) compose une expression régulière égale à l'une ou l'autre des deux expressions élémentaires

ERE	correspond à
(a|zzz)c	*ac* ou *zzzc*
([abc])\{3\}	*abc* ou *acb* ou *bac* ou *bca* ou *cab* ou *cba*

II.2.2.

III.

III. Exemples d’utilisation de commandes

III.1. Quelques fonctions du KornShell

III.1.1.

III.1.1. cd, pwd

Les commandes `cd` et `pwd` sont des commandes internes au shell, qui, respectivement, positionnent et lisent la variable PWD dans l’environnement courant.

Pensez-y lorsque vous créez des fichiers de commandes ou des macro-commandes de type (), car il y création d’un environnement séparé et l’effet de la commande `cd` sera perdu au retour, en même temps que l’environnement.

```
$ cd /usr/tmp
$ echo $PWD $OLDPWD
/usr/tmp /users/logis/doc/manuell_shell/exemples
$ pwd
/usr/tmp
$ cd -
/users/logis/doc/manuell_shell/exemples
$
```

La variable `OLDPWD` contient le répertoire courant précédant la dernière commande `cd` ; on peut retrouver le précédent répertoire en tapant `cd -` ; le répertoire courant est alors affiché automatiquement.

III.1.2. if

La commande if permet de créer dans les programmes des structures conditionnelles.

Principe :

```
if expression_conditionnelle
then
then instruction10
     instruction11
     ...
else instruction20
     instruction21
     ...
fi
```

III.1.3. La séquence `else` est optionnelle, et une séquence `if - then - else - fi` ou `if - then - fi` est considérée comme une instruction, ce qui permet d'imbriquer les tests ; mais attention : il devra y avoir toujours autant de `if` que de `fi`.

III.1.3. for

La commande `for` permet de réaliser des boucles avec un nombre d'itérations fini, déterminé par le nombre d'arguments donnés au `for`.

Attention : son fonctionnement n'est pas celui habituellement rencontré dans les langages de programmation courant !

Principe :

for *var* in *para*1 *para*2 *para*3 ...

`do`

instruction1

instruction2

...

`done`

La boucle est constitué des instructions (commandes shell, fonctions, etc) situées entre le `do` et le `done`. Avant chaque exécution de la boucle, la variable *var* est instanciée avec un nouveau paramètre *para*, et la boucle est parcourue autant de fois qu'il y a de paramètres *para*.

La séquence `for - do - done` est considérée elle-même comme une instruction, ce qui veut dire qu'on peut imbriquer plusieurs boucles.

```
$ for fichier in toto*
> do
> print $fichier
> done
toto1.c
toto2.c
toto_base.c
$
```

Ici, les paramètres sont tous les noms de fichiers du répertoire courant dont le nom commence par "toto", après transformation par le shell du terme `toto*`.

III.1.4. la commande "point" .

Un fichier de commande ou un programme shell est démarré dans un environnement séparé, ce qui veut dire qu'un fichier de configuration ne peut pas être lancé de manière classique sur la ligne de commande, puisqu'en fin d'exécution, l'environnement local qui vient d'être positionné est perdu.

Pour éviter cela, on lance le fichier de configuration par l'intermédiaire de la commande “point”, qui conserve l'environnement du shell courant.

```
$ more point
# positionnement d'une variable
variable=COUCOU_TRALALA
$ echo $variable

$ point
$ echo $variable

$ . point
$ echo $variable
COUCOU_TRALALA
$
```

Si vous placez le positionnement de variables, d'alias et de fonctions dans un fichier nommé *config_perso* par exemple, vous pourrez configurer le shell avec votre configuration par la commande :

`.` *config_perso*

ou `.` */chemin_absolu/config_perso*

si le fichier n'est pas dans le répertoire courant.

Si vous voulez que chaque shell démarré dans une fenêtre ait automatiquement cette configuration, vous pouvez mettre cette commande dans votre fichier `.kshrc`[1].

III.1.5. print

La commande `print` a les même fonctionnalités que la commande `echo`, mais :

• `print` est interne au shell alors que `echo` est une commande UNIX (binaire exécutable) : il est moins coûteux d'utiliser `print` que `echo`.

• `print` utilisé avec l'option `-u`*n* permet d'écrire directement dans le descripteur *n* ; vous serez obligé de manipuler les redirections pour obtenir le même résultat avec `echo`.

• `print` possédant une option, il vous sera impossible d'afficher une ligne dont le premier caractère commence par un tiret : par convention, le tiret désigne une option ; donc une ligne commençant par un tiret provoquera un message d'erreur du style : “`print : bad option(s)`”. Par contre `echo` n'a pas d'option, donc peut tout afficher.

1. Voir page 163

III.1.6.

III.1.6. set

La commande `set` permet de positionner certains paramètres de fonctionnement du shell ; en général, lorsque l'option est fournie avec un "moins", la caractéristique est validée, avec un "plus", elle est supprimée.

`set -A` *`tableau arg1 arg2 arg3 arg4 ...`*

supprime la variable *`tableau`* si elle existe, puis crée un tableau nommé *`tableau`* dont les éléments sont ceux fournis comme arguments.

Avec +A, la variable t*`ableau`* n'est pas supprimée en premier lieu.

`set -s` trie les paramètres de position $0, $1, $2 etc...

```
$ more pos_p
#
# donne les parametres de position avec
# lesquels ce programme a ete appele et les trie.

print $0,$1,$2,$3,$4
set -s
print $0,$1,$2,$3,$4

$ pos_p UN deux zzzzz coucou
pos_p,UN,deux,zzzzz,coucou
pos_p,UN,coucou,deux,zzzzz
$
```

`set -a` après l'exécution de cette commande, toutes les variables affectées sont exportées.

`set -u` lire une variable non initialisée provoque une erreur. Avec +u , une variable non initialisée est égale à une chaîne vide.

`set -v` la ligne qui va être exécutée, *avant* transformation, est affichée par le shell sur la sortie d'erreur. Pour repasser en mode normal, utiliser +v.

`set -x` la ligne qui va être exécutée, *après* transformation, est affichée par le shell sur la sortie d'erreur. Pour repasser en mode normal, utiliser +x.

Ces deux commandes sont intéressantes dans un programme pour fournir une trace d'exécution.

```
$ set -v
$ set -x
set -x
$ exec 1>/dev/null
exec 1>/dev/null
+ exec
+ 1> /dev/null
$ ls *
ls *
+ ls -FC essai1 essai2 pos_p resu_ls toto1.c toto2.c toto_base.c tutu tyty
$
```

Dans l'exemple ci-dessus, on positionne les options -v et -x pour avoir une trace de chaque commande, respectivement avant et après les transformations ; et pour se débarrasser du résultat qui importe peu ici, on redirige la sortie standard vers le fichier `/dev/null`.

Les lignes qui apparaissent avec un + sont celles générées suite à l'option -x ; la copie de la commande tapée (avant interprétation par le shell) est due à l'option -v.

On voit que la commande `ls *` a été transformée en deux temps :

- `ls` est devenu `ls -FC` : un alias existe donc vraisemblablement

- * est devenu la liste des fichiers dans le répertoire courant

III.1.7. shift

`shift` *n* décale de n rang à gauche, ou d'un rang si n est omis, les paramètres $1, $2, $3, etc positionné à l'appel d'un programme shell avec les paramètres fournis. Cette commande est intéressante lorsqu'on teste les arguments en début d'un programme[1].

III.1.8. trap

Permet d'attraper un signal ou de l'ignorer. Cette commande doit être placée en début de programme shell, avant réception du premier signal possible.

Pour attraper un signal :

`trap` *`commande signal1 signal2 signal3`* `...`

Sur réception d'un des signaux indiqués, un déroutement du programme est effectué et *`commande`* est exécutée. Si *`commande`* ne fait pas appel à `exit`, le programme reprend alors à l'endroit où il se trouvait au moment de la réception du signal.

1. voir le chapitre sur les programmes "propres" page 194.

III.1.8. *commande* peut être une commande UNIX, un fichier de commandes, une fonction...

Pour ignorer un signal :

`trap "" ` *signal1 signal2 signal3* `...`

Le programme qui a fait appel à cette commande n'est plus sensible à la réception des signaux indiqués.

Pour lister les signaux attrapés :

```
trap
```

Exemple 1 :

```
$ trap "print Reception du signal USR1" USR1
$ kill -USR1 $$
Reception du signal USR1
$
```

Les signaux USR1 et USR2 n'ont pas de signification particulière et sont entièrement disponibles pour les utilisateurs. La variable $$ contient l'identificateur du processus courant, c'est à dire le shell lui-même : le shell s'est donc envoyé un signal USR1 à lui-même et il réagit en affichant le message.

Exemple 2 :

Lorsque vous vous déconnectez, tous les processus lancés par le shell intéractif reçoivent un signal HUP, même ceux qui s'exécutent en background à ce moment là : ils sont tous tués.

Pour qu'une commande ne soit pas tuée si elle s'exécute et que vous vous déconnectez, vous pouvez :

- la lancer par l'intermédiaire de la commande `nohup`
- si c'est un programme shell, rajouter : `trap "" HUP`

La commande `nohup` ne fait rien de plus que d'ignorer le signal HUP avant de lancer la commande que vous lui avez fournie en argument.

La commande `trap` est très utile dans les programmes shell qui utilisent des fichiers temporaires : on peut dérouter le signal INT provoqué par un appui <control>C, et supprimer proprement ces fichiers avant de rendre la main[1] voir le chapitre consacré aux programmes shell "propres".

De la même façon, `trap` est intéressant avec deux signaux :

- DEBUG qui déroute après chaque commande

1. Voir le chapitre consacré aux programmes shell "propres" page 200

- EXIT qui déroute après l'exécution de la dernière ligne d'un programme shell: III.1.8.

```
 1 #!/bin/ksh
 2 # manipulation des traps
 3 #
 4 trap "echo FIN DU SCRIPT" EXIT
 5 trap 'echo ligne $LINENO: temps: $SECONDS' DEBUG
 6 # debut du programme ...
 7 SECONDS=0
 8 date
 9 sleep 3
 10 date
 11 sleep 5
 12 date
 13 exit 0
 14
~
~"toto" 14 lines, 192 characters
```

```
$ toto
ligne 7: temps: 0
Sat Jun 05 15:38:35 1993
ligne 8: temps: 0
ligne 9: temps: 3
Sat Jun 05 15:38:38 1993
ligne 10: temps: 3
ligne 11: temps: 8
Sat Jun 05 15:38:43 1993
ligne 12: temps: 8
FIN DU SCRIPT
ligne 13: temps: 8
$
```

III.1.9.

III.1.9. typeset

La commande typeset permet de typer les variables utilisées.

Le principe d'option avec + ou - est le même que pour `set` : - pour valider, et + pour invalider. L'affectation peut être faite en même temps (en rajoutant =*`valeur`* juste à la suite)

`typeset` -L *`variable`*

`typeset` -L*`n variable`*

justifie à gauche la `variable`. Si n est donné, la largeur de la `variable` est imposée à *n* caractères, éventuellement complétée par des blancs ; sinon elle vaut la largeur lors de la première affectation. Les blancs en début de champ sont ignorés.

Pour profiter de la justification, il est nécessaire lorsqu'on *emploie* une variable justifiée de l'encadrer avec des double-quotes.

```
$ typeset -L8 toto
$ toto=1
$ echo $toto FIN
1 FIN
$ echo "$toto FIN"
1        FIN
```

`typeset` -R *`variable`*

`typeset` *`-Rn variable`*

idem précédent, mais la justification se fait à droite.

`typeset` -Z *`variable`*

`typeset` –Z*`n variable`*

justifie à droite et complète avec des zéros si nécessaire.

`typeset -i`*`base variable`*

la variable est de type entier en base `base`. Si `base` est omis, la valeur par défaut est la base 10.

On peut alors écrire directement (et sans espace) :

`variable``=`*`variable`*`+1`

pour incrémenter une variable de type entier.

III.1.9.

En effet, seules les variables de type entier peuvent être affectées directement avec des expressions arithmétiques.

```
$ typeset -i2 val
$ val=(8*4+1)
ksh: syntax error: `(' unexpected
$ val="(8*4+1)"
$ echo $val
2#100001
$ echo "(8*4+1)"
(8*4+1)
```

On voit dans l'exemple qui précède que les expressions arithmétiques ne sont évaluées que s'il y affectation dans une variable de type entier ; sinon, l'expression est prise comme une chaîne. Il se peut que le shell pose problème avec certains caractères de l'expression : dans ce cas, on l'encadre avec des double-quotes.

Lorsque la base est différente de 10, son indication à l'affichage est explicite, sous la forme suivante: `base#nombre`. Dans l'exemple ci-dessus, on affecte la variable `val` typée en *entier base 2* avec la valeur 33 (8*4+1), et on obtient lors de sa lecture `2#100001`, c'est à dire le nombre 100001 en base 2. (Voir ci-dessous comment enlever l'indication de la base)

`typeset -r` *`variable`*

la variable est de type *read-only* : on ne peut plus la modifier.

`typeset -u` *`variable`*

à chaque affectation, le contenu de la variable est passé automatiquement en majuscules.

`typeset -l` *`variable`*

idem en minuscules.

Il est possible de changer de type une variable déjà typée :

```
$ typeset -R3 val=123456
$ echo $val
456
$ typeset -i val
$ val=val*10+1
$ echo $val
4561
```

III.1.9. ou de faire des conversions de type :

```
$ typeset -u chaine="ToTo" ; echo $chaine
TOTO
$ typeset -l chaine ; echo $chaine
toto
$ typeset -i2 entier=255 ; echo $entier
2#11111111
$ typeset -i16 entier ; echo $entier
16#ff
$ echo ${entier#*#}
ff
```

pour se débarrasser de la valeur de la base quand on veut afficher des entiers dans une base autre que 10, on peut utiliser les opérateurs sur les variables qui permettent de supprimer les premiers caractères (la base et le #) : c'est la dernière commande de l'exemple ci-dessus.

Il est possible également de manipuler des constantes dans une base quelconque :

```
$ typeset -i decimal1
$ decimal1="2#101010101010"
$ echo $decimal1
2#101010101010
$ typeset -i10 decimal2
$ decimal2="2#101010101010"
$ echo $decimal2
2730
```

Ci-dessus, on constate qu'une variable de type entier mais sans indication de base, adopte la base de la première valeur affectée.

Par contre, si on spécifie la base au départ, une conversion est effectuée automatiquement.

Une autre utilisation de `typeset` permet de lister les fonctions enregistrées dans le shell courant :

```
typeset -f
```

ou les variables de type entier :

```
typeset -i
```

etc ...

III.1.10. unset

III.1.10.

`unset` *`variable`*

Cette commande permet de supprimer la variable *`variable`* de l'environnement (exporté ou non).

III.1.11. wait

`wait` *`job`*

Cette commande permet de synchroniser le shell ou programme courant avec un processus s'exécutant en arrière-plan, et lancé par le shell ou le programme courant.

La commande `wait` se bloque jusqu'à ce que le *`job`* soit terminé, ou si *`job`* est omis, que tous les processus qui tournent en arrière-plan soient terminés.

`job` peut prendre la forme :

%des	avec *des* descripteur de processus (donné par le shell au moment du lancement)
pid	identificateur (numéro du processus pour le système, donné par `ps`)

Vous pouvez ainsi lancer en parallèle plusieurs traitements secondaires en arrière-plan depuis votre programme principal, puis attendre leur fin, avant de continuer les calculs ; dans l'exemple ci-dessous, la macro-commande () symbolise le traitement secondaire.

```
$ ( sleep 100 ; echo fin du process1 ) &
[1] 7181
$ ( sleep 100 ; echo fin du process2 ) &
[2] 7183
$ wait ; print "Reprise d'execution"
fin du process1
fin du process2
Reprise d'execution
[2] + Done ( sleep 100 ; echo fin du process2 ) &
[1] + Done ( sleep 100 ; echo fin du process1 ) &
$
```

III.1.12.

III.1.12. whence

`whence -v` *`command`*

Indique comment le shell va interpréter le terme *command* si celui-ci est supposé être une commande. Dans le cas où un fichier exécutable correspond à *`command`*, le chemin complet est donné, si *`command`* est un alias ou une fonction, `whence` l'indique aussi.

Si plusieurs possibilités existent (un terme peut correspondre à la fois à un alias, une fonction et plusieurs fichiers exécutables trouvés dans le PATH), `whence` ne donne que celle qui est prise en considération au moment de l'exécution de *`command`*.

III.1.13. while

La commande `while` permet de réaliser des boucles avec un nombre d'itérations indéterminé.

Principe :

```
while expression_conditionnelle
do
    instruction1
    instruction2
    ...
done
```

Tant que l'*`expression`* est vraie, c'est à dire que son code de retour vaut zéro, la boucle constituée des instructions comprises entre le `do` et le `done` est exécutée cycliquement. Le type de l'*`expression`* est celui indiqué pour la commande conditionnelle `if` (voir page 82).

Comme c'est le code de retour qui est testé, rien n'empêche de remplacer *`expression_conditionnelle`* par une commande : tant que l'exécution de la commande provoque un code de retour nul, le `while` continue à boucler.

La séquence `while-do-done` étant considérée elle-même comme une instruction, il est possible d'en imbriquer plusieurs.

III.2. Quelques commandes UNIX III.2.1.

Avertissement

Ce chapitre n'est pas une référence sur l'utilisation des commandes et de leurs options : il est destiné uniquement à en mieux maîtriser quelques-unes des plus courantes.

C'est pourquoi il ne dispense pas de jeter un coup d'oeil sur le manuel de référence(manuel(1)) qui les détaille de manière exhaustive.

III.2.1. awk

But

Travailler le contenu d'un fichier texte

Le mode naturel de manipulation de `awk` consiste à décomposer la ligne courante en champs (contrairement à `sed` qui travaille caractère par caractère).

Version

Attention à la version du `awk` dont vous disposez : certaines versions sont moins riches, bien que le fonctionnement soit globalement équivalent.

voir

voir

```
$ what $(whence awk)
/usr/bin/awk:
 $Revision: 66.49 $
$
```

Structure du programme

`awk` permet d'écrire des programmes complets, avec structures itératives, tests, etc...

Le programme est passé comme paramètre (encadré avec des quotes) à `awk`, ou écrit dans un fichier dont le nom est fourni par l'intermédiaire de l'option `-f`.

Exemple :

```
awk '/ACE/ { print $1, $3 }' mon_fichier
```

Dans l'exemple ci-dessus, le programme fourni à `awk` est constitué par la chaîne

```
'/ACE/ { print $1, $3 }'
```

et s'exécute sur *`mon_fichier`*.

III.2.1. Structure générale :

BEGIN{ *action_begin* **}**
sélection_1 **{** *action_1* **}**
sélection_2 **{** *action_2* **}**
...
END{ *action_end* **}**

Le fonctionnement est le suivant :

- avant de commencer à lire *mon_fichier*, awk exécute *action_begin*
- pour chaque ligne de *mon_fichier*, awk évalue successivement les *sélections* du programme : si la sélection est valide pour la ligne courante, l'*action* correspondante est exécutée.

  Si plusieurs *sélections* sont valides pour une ligne de *mon_fichier*, alors plusieurs *actions* (celles correspondantes) seront exécutées
- après lecture complète de *mon_fichier*, awk exécute *action_end*.

Structure modulaire

Aucun des éléments indiqué ci-dessus n'est indispensable lorsqu'on écrit un programme awk.

- On peut omettre la séquence `BEGIN{...}`, aussi bien que celle `END{...}`.
- Si on omet *sélection_1*, alors { *action_1* } est exécutée pour toutes les lignes de *mon_fichier*.
- Si on omet { *action_1* }, alors les lignes pour lesquelles *sélection_1* est valide sont recopiées in-extenso sur la sortie standard.

Exemple :

`awk '/ACE/ { print $1, $3 }'` *mon_fichier*

* pas de séquence BEGIN{ }

* pas de séquence END{ }

* une sélection : `/ACE/`

l'action correspondante : `{ print $1, $3 }`

Résultat : toutes les lignes qui contiennent la chaîne ACE ont leur premier et troisième champs imprimés sur la sortie standard.

Séparation des champs

Par défaut, les champs sont définis comme une suite de caractères ne comprenant ni <espace>, ni <tabulation>.

Mais il possible d'utiliser d'autres caractères de séparation de champs que ceux par défaut, et même de définir le séparateur de champs avec une expression régulière.

Pour ce faire on utilise l'option -F*séparateur* de la commande awk, ou alors on positionne la variable FS dans la séquence BEGIN{ } du programme.

III.2.1.

Exemple 1 :

On considère le séparateur de champs : /

```
$ echo "05/11/92" |
> awk -F/ '{ print $3}'
92
$
```

Exemple 2 :

On considère le séparateur de champs constitué de la chaîne :

`val=`

précédée ou suivie par des caractères <espace>, <tabulation>[1] ou <virgule>

voir IBM

```
$ more sep_awk
val=3, val=4,val=5
110 val=2, val=5
val=3 , val= "valeur"
val=6, val=9
$ awk 'BEGIN{ FS = "[ \t,]*val=[ \t,]*" }
> { $1 = $1 ; print }' sep_awk
 3 4 5
110 2 5
 3 "valeur"
 6 9
$
```

On remarque que certaines lignes du fichier résultat commencent par un caractère blanc : ce sont celles qui correspondent dans le fichier de départ aux lignes débutant par "val=". Ce terme étant le séparateur, et se trouvant en début de ligne, on retrouve un espace en début de ligne dans le fichier de sortie.

La commande ci-dessus revient à faire une conversion du séparateur "val=" en un séparateur <espace> en sortie. Or, si aucune manipulation n'est faite sur un des champs de la ligne d'entrée, `awk` ne réalise pas cette conversion, et recopie les lignes dans l'état où elles sont en entrée. C'est pourquoi le petit programme `awk` précédent contient l'instruction "$1 = $1" qui n'a ici aucune utilité sinon de manipuler un champ de la ligne et donc de forcer la conversion des séparateurs.

Variables internes

`awk` possède des variables internes positionnées dynamiquement à la lecture de chacune des lignes de *`mon_fichier`*.

1. symbolisé par la séquence : **\t**

III.2.1. Parmi les plus intéressantes :

NF : le nombre de champs dans la ligne

NR : le numéro de la ligne courante dans *mon_fichier*

ENVIRON["*var*"] : le contenu de la variable shell *var* exportée avant appel de `awk`

$*i* : le *i*ème champ de la ligne.

Il est possible de créer autant de variables que l’on veut, ou de modifier celles existantes dans les *actions*.

Remarques :

Si la variable A vaut 3, alors $A est le troisième champ de la ligne.

De la même façon, $NF est le dernier champ de la ligne, et $(NF-1) est l’avant dernier champ de la ligne.

Bien sûr, il ne vous viendrait pas à l’idée de confondre la notation :

- $NF pour le shell, qui vaut le contenu de la variable shell NF,

avec :

- NF pour le `awk`, qui vaut le contenu de la variable `awk` NF,

et avec :

- $NF pour le `awk`, qui vaut le contenu du champ dont le numéro est donné par la variable NF.

Attention: dans un programme `awk`, toute chaîne de caractères non encadrée par des double-quotes est considérée comme étant le nom d’une variable, (et si cette variable n’a jamais été affectée, une chaîne vide est renvoyée lors de sa lecture).

Description d’une sélection

La sélection peut prendre principalement deux formes :

Une expression régulière :

* sous la forme : /*expression*/

dans ce cas, toutes les lignes qui vérifient l’expression régulière *expression* vérifient la sélection.

* sous la forme : $*i* ~ /*expression*/

dans ce cas, la sélection est vérifiée si le champ *i* vérifie l’expression régulière *expression*.

Exemple :

$1 ~ /^[0-9]/

cette sélection est vérifiée quand l’expression régulière `^[0-9]`[1] est vérifiée sur le premier champ de la ligne.

1. elle signifie : le premier caractère (^) est un chiffre ([0-9])

III.2.1.

Une relation :

Avec les opérateurs < <= == != >= >

Exemples :

$1 == "toto"

vrai si le premier champ est égal au terme toto

$3 > "toto"

vrai si le troisième champ est supérieur alphabétiquement au terme toto

NR == 10

vrai si la variable NR est égale à 10 (si le numéro de la ligne courante est égal à 10)

Combinaison des sélections

On peut combiner des sélections avec les opérateurs :

&&	ET logique
||	OU logique
!	NON logique
( )	association
,	définition d'une séquence de bloc :

les deux sélections séparées par une virgule déterminent respectivement la ligne de début et celle de fin de bloc.

Toutes les lignes du bloc, y compris celle de début et de fin sont sélectionnées.

Exemple :

```
/begin/,/end/
```

sélectionne dans le fichier tous les blocs de lignes commençant par une ligne contenant "begin", et finissant par une ligne contenant "end".

Exemples de sélections :

NF == 3

les lignes contenant exactement trois champs sont sélectionnées.

```
/debut/ && $NF !~ /^[0-9]/
```

les lignes qui contiennent "debut" et dont le dernier champ ne commence pas par un chiffre.

```
( /un/ && /deux/ ) || /trois/
```

les lignes qui contiennent "un" et "deux", ou les lignes qui contiennent "trois".

```
! /exemple/ && NR >10
```

les lignes qui ne contiennent pas "exemple", et dont le numéro à partir du début du fichier est supérieur à 10.

III.2.1.

Description d’une action

Une action ressemble en gros à une portion de code en langage C.

Cependant, un programme `awk` est interprété, ce qui évite ici d’avoir à déclarer les variables utilisées : une variable non déclarée est toujours nulle ou vide à la première utilisation.

Toute constante de type chaîne apparaissant dans un programme awk doit être encadrée par des double-quotes.

Les opérateurs de calcul sont : **+**, **-**, *****, **/**, **%**[1] et **^**[2]

Lors d’une affectation, on peut concaténer des champs en les séparant par des <espace> :

```
$ awk 'END{ a = "toto" "tutu"; print a }' /dev/null
tototutu
$
```

Dans l’exemple ci-dessus, on ne veut pas travailler avec un fichier d’entrée : on utilise alors `/dev/null` qui est toujours vide, et la séquence END{ } du programme `awk`.

Fonctions internes

Il existe des fonctions internes mathématiques (cos, sin, sqrt,...), et de manipulation de champs.

Les plus intéressantes sont celles de manipulation de champs et d’impression :

`print`

Permet d’imprimer un résultat.

Chaque commande print provoque un retour à la ligne.

Sans paramètre, la ligne courante du fichier d’entrée est recopiée telle quelle.

Avec des paramètres séparés par des blancs, les paramètres sont imprimés concaténés sur la sortie standard.

1. modulo
2. exponentielle

III.2.1.

Lorsque les paramètres du `print` sont séparés par des virgules, ils sont imprimés séparés par des <espace> ; si les paramètres sont séparés par des <espace>, ils sont imprimés concaténés.

```
$ echo "12345 7 9" |
> awk '{ print ;
>        print $1,$2 ;
>        print $1 $2 }'
12345 7 9
12345 7
123457
$
```

On peut envoyer le résultat dans un autre fichier que la sortie standard :

```
print $1, $2, $3 >"resultat"
```

envoie les champs un, deux et trois de la ligne courante dans le fichier "`resultat`".

`printf`

Idem la commande print, mais la sortie est formatée.

Cette fonction travaille de la même manière qu'en langage C.

(le premier paramètre est impérativement une chaîne contenant le format des paramètres suivants)

voir IBM

```
$ echo "123.456 111 255 16 un deux" |
> awk '{ printf "%3.2f %d %X %X", $1, $2, $3, $4;
>        printf "%5s %s\n", $5, $6 }'
123.46 111 FF 10    un deux
$
```

Il y a deux commandes `printf` : la première imprime les 4 premiers champs de la ligne d'entrée, et la seconde, les 2 champs restant.

La commande `printf` ne réalise un retour à la ligne uniquement si la séquence **\n** est spécifiée dans le format (c'est le cas ici dans la deuxième commande seulement, ce qui fait que la sortie se trouve finalement sur une seule ligne).

III.2.1. Schéma du formatage :

```
$ echo "123.456 111 255 16 un deux" |

> awk '{ printf "%3.2f %d %X %X", $1, $2, $3, $4 ;

              123.46 111 FF 10    un deux
               3  2
          $ echo "123.456 111 255 16 un deux" |

>         printf "%5s %s\n", $5, $6 }'

123.46 111 FF 10    un deux
$
```

5 caractères prévus

Dans l'exemple donné, on repère les principaux symboles représentant le type des données à imprimer :

- `%f` pour flottant ;
- `%d` pour décimal
- `%X` pour hexadécimal majuscule (avec conversion)
- `%s` pour chaîne ascii

Le nombre apparaissant parfois est une indication de la longueur du champ pour le formatage.

On peut avoir plus d'indications sur cette commande en consultant le manuel(3) à la rubrique `printf`.

sprintf

Idem `printf`, mais le résultat n'est pas envoyé sur la sortie standard: cela permet d'affecter des variables.

```
$ echo "123.456 12 12.123456" |
> awk '{ res = sprintf("%3.2f %3.2f %3.2f",$1,$2,$3);
> print "res=", res }'
res= 123.46 12.00 12.12
$
```

III.2.1.

length

Donne la longueur totale de la ligne, ou d’un champ si on le fournit en argument.

```
$ echo "12345 789" |
> awk '{ print length, length($1) }'
9 5
$
```

Sans argument, on a la longueur totale de la ligne (ici 9 caractères) ; ensuite, la longueur du premier champ ($1) (ici, 5 caractères).

index

Donne la position d’une sous-chaîne à l’intérieur d’une chaîne.

```
$ echo "T_SPE_CTXP_DF1" |
> awk '{ print index($1,"SPE") }'
3
$
```

match

Idem précédent, mais on peut spécifier une expression régulière à rechercher plutôt qu’une chaîne fixe.

substr

Extraction d’une partie de chaîne

```
$ echo "T_SPE_CTXP_DF1" |
> awk '{ print substr($1,7,4) }'
CTXP
$
```

(extrait 4 caractères à partir de la position 7 du premier champ)

sub

Substitution

```
$ echo "T_SPE_CTXP_DF1" |
> awk '{ sub("DF[012]","DFL",$1); print $1 }'
T_SPE_CTXP_DFL
$
```

III.2.1. Le premier paramètre de `sub` est une expression régulière étendue[1], le second, le terme de remplacement, le troisième, la variable à modifier.

La fonction renvoie 1 si la modification s’est bien passée, 0 sinon.

La modification n’est faite que sur le premier terme trouvé correspondant à l’expression régulière.

gsub

Idem précédent mais toutes les occurrences de l’expression régulière sont modifiées (pas seulement la première rencontrée).

split

Décompose en plusieurs éléments et affecte un tableau.

```
$ echo "T_SPE_CTXP_DF1" |
> awk '{ n = split($1,tab,"_") ;
>        print "nombre de sous-champs:", n ;
>        for(i=1 ;i<=n ;i++)
>           print "No",i,"=",tab[i] ;
>      }'
nombre de sous-champs: 4
No 1 = T
No 2 = SPE
No 3 = CTXP
No 4 = DF1
$
```

Ici, on décompose le premier champ ($1) en sous-champs (séparés par des "_") dans un tableau nommé "tab".

Au passage, on voit comment faire une boucle `for` (comme en C).

toupper

tolower

Convertit respectivement en majuscules et en minuscules.

Vous pouvez, par exemple, faire un test sur un champ indépendamment de la taille de caractères :

`tolower($1) == "toto"`

teste si le premier champ vaut "toto" ou "TOTO" ou "ToTo", etc...

Commentaires

Vous pouvez placer des commentaires dans le programme derrière le caractère **#**.

1. attention aux caractères réservés comme (,), + etc. des expressions régulières; voir page 76.

Les commentaires doivent se trouver à l'intérieur d'une séquence `{ }`, c'est à dire qu'il n'est pas possible d'en mettre entre le corps `BEGIN{}` et le corps principal, ou alors entre le corps principal et le corps `END{}`. III.2.1.

Attention: si votre programme `awk` est placé entre quotes sur la ligne de commande, comme cela est fait dans les exemples qui précèdent, il n'est pas question d'insérer une apostrophe (apostrophe = quote) dans le texte, même si c'est du commentaire: si vous faites cela, votre apostrophe sera considérée comme la quote de fin de programme, et vous aurez une erreur de syntaxe à l'exécution. Par contre, si votre programme est placé dans un fichier (option `-f` de `awk`), il n'est alors pas encadré avec des quotes, et rien ne vous empêche de mettre des apostrophes dans vos commentaires.

Passage à la ligne

Vous pouvez couper les lignes du programme, par exemple après les caractères <point virgule>, <virgule>, **&&** , **||**, et entre les mots clé (`if`, `then`, `for`...), mais pas au milieu d'une chaîne (encadrée par des double-quotes `"`).

Exemples complets de programmes awk

Exemple 1 : total sur une colonne d'un fichier

On veut connaître la taille globale des fichiers suffixés en .c dans le répertoire courant.

La taille en octets d'un fichier est le cinquième champ dans le résultat d'une commande `ls -l` ; d'où :

```
$ ls -l *.c
-rw-r--r-- 1 logis users 84 Apr 15 15:38 toto1.c
-rw-r--r-- 1 logis users 84 Apr 15 15:39 toto2.c
-rw-r--r-- 1 logis users 92 Apr 15 15:39 toto_base.c

 $ ls -l *.c |
 > awk '{ total += $5 ;
 >        printf "%14s: taille = %d \n", $NF, $5 ;
 >       }
 >       END{ print "Total = " total }'
         toto1.c: taille = 84
         toto2.c: taille = 84
     toto_base.c: taille = 92
 Total = 260
 $
```

Explication :

- Pour toutes les lignes, on somme dans la variable `total` la taille du fichier ($5), et on imprime le nom du fichier ($NF) avec la taille correspondante ($5). Pour garder les noms des fichiers alignés en sortie quelque soit leur longueur, on utilise la fonction `printf` pour le formatage.
- Une fois toutes les lignes lues (séquence END{ }), on imprime la variable `total` qui contient le résultat.

III.2.1. Exemple 2 : idem précédent mais suivant plusieurs critères

On recherche par exemple le total pour chacun des groupes (au sens UNIX) des fichiers se trouvant dans le répertoire `/tmp`.

La base est la même que précédemment, mais il faut tenir un total pour chacun des groupes rencontrés ; (le groupe est le 4ème champ dans un listing de la commande `ls -l` ; dans l'exemple au dessus, ce serait "users").

Cette fois, on n'affichera que les totaux, et non plus les valeurs pour chacun des fichiers.

```
$ cd /tmp
$ ls -l |
awk 'NF>3 { tabtot[$4] += $5 ;
               total += $5 ;
          }

     END{
          for ( i in tabtot )
             printf "total %6s = %7d\n",i,tabtot[i];
         print "\nTotal general = " total ;
        }'
total root   =   11532
total ML2    =  865864
total sys    =     993
total iliade =  273408
total CMFR   = 1203305

Total general = 2355102
$
```

Explications :

La structure du programme est la suivante :

sélection { *action1*}

END{ *action2* }

* La sélection : NF > 3

les totaux (action1) ne sont réalisés que si le nombre de champs dans la ligne est supérieur à 3 ; le résultat de la commande `ls -l` fournit en première ligne le total en nombre de bloc (512 octets) du répertoire listé ; cette ligne ne contient que 2 champs : "total" et la valeur en nombre de blocs. En ne gardant que les lignes ayant plus de 3 champs, on ignore donc cette ligne indésirable.

* *action1*

on crée un tableau dont l'indice est le nom du groupe rencontré : de cette manière, on n'est pas obligé de connaître le nombre ou la liste des groupes qui peuvent exister. C'est la démonstration que les tableaux acceptent comme indice une chaîne de caractères, et non pas seulement un entier.

Si le groupe est ML2 par exemple, l'affectation est alors :

III.2.1.

```
tabtot["ML2"] += xxx
```

avec xxx égal à la valeur du 5ème champ pour cette ligne.

* *action2*

Ceci correspond à l'affichage des résultats.

Telle qu'elle est écrite, la boucle `for` liste tous les éléments du tableau `tabtot` sans avoir besoin d'en connaître le nombre.

Conclusion :

La manipulation des tableaux par `awk` est très puissante, et permet une grande souplesse d'utilisation.

Exemple 3 : mise sur une ligne de données trouvées sur des lignes successives

La transformation consiste à passer d'un fichier de la forme suivante :

```
nom     val1
prenom  val2
age     val3
nom     val4
...
```

à un fichier de la forme :

```
val1  val2  val3
val4  ...
```

Voici un exemple de programme `awk` dans le cas le général où les lignes "nom", "prenom", "age" apparaissent dans l'ordre mais ne sont pas toutes les trois toujours présentes (il peut y avoir un saut de "nom" à "age" avec omission de "prenom" par exemple).

III.2.1.

Notez qu'il existe des solutions beaucoup plus simples quand le format du fichier est très rigide ; mais ce n'est pas le cas ici, et on découvre ainsi la souplesse de `awk` pour traiter des cas complexes :

```
$ more convers.awk
# --- programme AWK ---
# passage de donnees reparties dans plusieurs
# lignes en une seule ligne

BEGIN{          info1 = info2 = info3 = "-" ; }

               { nr = 0 }
$1 == "nom"    { info = $2 ; nr = 1 ; }
$1 == "prenom" { info = $2 ; nr = 2 ; }
$1 == "age"    { info = $2 ; nr = 3 ; }
               {
# la derniere info de la serie 1 2 3 est atteinte
                 if ( (nr <= old_nr) && (nr != 0) ) {
                      printf "%10s %8s %2s\n", \
                      info1, info2, info3 ;
                      info1 = info2 = info3 = "-" ;
                      old_nr = nr ; reset = 0 ;
                  }
                }
nr > old_nr     { old_nr = nr }
nr == 1         { info1 = info ; reset = 1 }
nr == 2         { info2 = info ; reset = 1 }
nr == 3         { info3 = info ; reset = 1 }

END{              if ( reset == 1 )
                     printf "%10s %8s %2s\n", \
                             info1, info2, info3
}
```

Le principe est le suivant :

Chaque type d'information est associé à un numéro croissant en fonction de sa position prévue dans la suite : "nom", "prenom", "age".

On repère le type de la ligne courante, on élimine les commentaires et les lignes blanches. Normalement, le type de l'information courante doit être de rang plus élevé que celui de la ligne précédente ; sinon, c'est qu'on change d'enregistrement.

Dans ce cas, on imprime une ligne avec les informations recueillies dans les lignes successives du fichier d'entrée.

Et voici un exemple d'exécution avec un fichier contenant les informations "nom", "prenom" et "age", dans l'ordre, pour plusieurs personnes. Les informations sont incomplètes. III.2.1.

```
$ more convers.input
# M Dubois
   nom Dubois
prenom jules
age 12
age 12

# M Dupond Albert
nom Dupond
age 52

nom Dupont
nom Pierre
$ awk -f convers.awk convers.input
    Dubois    jules  12
         -        -  12
    Dupond        -  52
    Dupont        -   -
    Pierre        -   -
$
```

On voit également ci-dessus la méthode permettant de placer le programme `awk` dans un fichier séparé.

Remarque

Une erreur courante lorsqu'on écrit des programmes `awk` est d'oublier d'encadrer les constantes de type “chaîne de caractères” entre des guillemets. Une chaîne de caractères non encadrée par des guillemets est considérée comme une variable, qui plus est, nulle, car il n'est pas indispensable de faire des déclarations.

Par exemple, si vous écrivez le test :

```
$1 == bind
```

alors, `bind` est une variable, qui, si elle n'a pas été déjà affectée, est considérée comme une chaîne nulle : le test sera toujours faux (si le premier champ existe, il ne pas être vide)

alors que pour

```
$1 == "bind"
```

`"bind"` est une chaîne de caractères de valeur : `bind`

III.2.2.

III.2.2. basename, dirname

But

Donne respectivement le nom du fichier et son chemin lorsqu’on applique ces deux commandes sur un chemin absolu de fichier.

Exemple

```
$ basename /usr/include/signal.h
signal.h
$ dirname /usr/include/signal.h
/usr/include
$
```

III.2.3. bs

voir

But

bs est un interpréteur-compilateur d’un langage se rapprochant du basic.

L’intérêt de bs est qu’il possède la plupart des fonctions mathématiques de base, et qu’il calcule en virgule flottante.

D’autre part, il permet de compiler des parties de programmes, et donc de faire des calculs numériques complexes de manière efficace.

Utilisation

bs s’utilise normalement en mode intéractif, mais il est aussi possible d’envoyer la série des ordres et des données passés dans son entrée standard.

Les exemples présentés ici sont triviaux. Pour le détail (qui est très fourni), se reporter au manuel(1) des commandes.

Exemples III.2.3.

en mode intéractif :

```
$ bs
2 * 3 + 0.5
6.5
u = 18 * log(2) * 1e-2
u
0.1247664925
obase 8
12
14
q
$
```

Lorsqu’on tape une expression, elle est évaluée, et son résultat éventuel est affiché à la ligne suivante.

`obase 8` signifie base 8 en sortie (à l’affichage).

Il n’est possible de manipuler que les bases 8, 16 et 10.

`q` pour sortir

en mode “piloté”

```
$ echo "rand()" | bs
0.5695495605
$ echo "floor(rand()*100)" | bs
17
$ echo "floor(rand()*100)" | bs
23
$
```

Ici on utilise quelques-unes des fonctions du langage (pseudo-basic) :

rand générateur de nombres aléatoires

floor partie entière.

Remarques

`bs` manipulant des variables, seuls les champs commençant par des chiffres sont considérés comme des nombres ; les autres sont supposés être des variables numériques nulles, si elles n’ont pas encore été initialisées.

Par exemple :

123 vaut 123

FFF vaut 0 , même en base 16 (c’est censé être une variable)

0f vaut 15 en base 16

D’autre part, les lettres dans les nombres hexa doivent apparaître en minuscules.

III.2.4.

III.2.4. dc

But

Permet entre autres de faire du calcul sur des nombres entiers, et des conversions d'unité.

Utilisation

`dc` s'utilise normalement en mode interactif, mais il est aussi possible d'envoyer la série des ordres et des données passés dans son entrée standard.

Comme `dc` travaille par pile, la syntaxe est de type notation inversée.

Pour les calculs, on lui préférera `bs`.

Exemples

en mode interactif :

```
$ dc
2o
16i
FF
p
11111111
q
$
```

`2o`	signifie base 2 en sortie.
`16i`	signifie base 16 en entrée.
`p`	est la commande d'impression du haut de la pile
`q`	pour sortir

en mode "piloté"

```
$ echo "1 2 + 3 * p" | dc
9
$
```

Remarque :

Les lettres dans les nombres hexa doivent apparaître en majuscules.

III.2.5. file

III.2.5.

<u>But</u>

Déterminer le type et le contenu d’un fichier.

<u>Utilisation</u>

`file` *`mon_fichier`*

<u>Principe</u>

Il existe pour le système UNIX une méthode d’identification du type des fichiers qu’il manipule : le test du *magic number*. Ce numéro fait partie des attributs du fichier[1] , et le caractérise : il peut indiquer un fichier binaire exécutable, un fichier texte, un module relogeable (.o), etc..

Lorsque le fichier est de type texte, l’éventail des possibilités reste large, et la commande `file` étudie les 500 premiers caractères pour essayer d’identifier le type du contenu. Il ne faut pas s’attendre à des miracles dans cette situation, sauf si le fichier est fortement typé : un programme shell commençant par `#!/bin/ksh` par exemple... Le contre exemple classique est un fichier contenant un programme `awk`, que la commande `file` confond le plus souvent avec un programme C.

Dans le cas où le fichier est de type texte, il est de toute manière bien plus simple de l’éditer directement que d’utiliser la commande `file` ; dans le cas de fichiers binaires, on n’a pas tellement le choix...

<u>Types reconnus</u> :

voir

voir

English text	texte (ascii)
[nt]roff, tbl, or eqn input text	idem
ascii text	...
assembler program text	...
awk program text	...
basic program text	...
c program text	...
commands text	...
fortran program text	...
lex command text	...
pascal program text	...
s200 assembler program text	...
s500 assembler program text	...
s800 assembler program text	...
sed program text	...

1. dans l’inode (voir le paragraphe sur les liens physiques: page 27)

III.2.6.

yacc program text	...
s200 pure executable	binaire

III.2.6. find

But

Le but principal de `find` est de lister les fichiers et répertoires d'une arborescence complète à partir d'un répertoire donné.

Exemple :

```
find /users -print
```

liste l'ensemble des fichiers et répertoires qui se trouvent sous `/users`, ainsi que tous ceux placés dans les sous-répertoires (ainsi de suite ...)

```
find . -print
```

fait de même pour le répertoire courant.

Filtres

L'intérêt majeur du `find` est qu'il est possible de rajouter des filtres pour ne conserver que certains fichiers ou répertoires ayant des caractéristiques particulières.

Exemple : filtre sur le nom de fichier

```
find /users -name '*.c' -print
```

explore récursivement `/users` mais ne conserve à l'affichage que les fichiers ayant un suffixe `.c`

On peut mettre plusieurs filtres pour affiner la recherche.

Exemple : filtre sur le nom et filtre sur la date de dernière modification

```
find /users -name '*.c' -mtime -3 -print
```

explore `/users` et ne conserve que les fichiers de suffixe `.c` modifiés il y a moins de trois jours.

Enchaînement des filtres

Lorsqu'il y a plusieurs filtres, l'interprétation est la suivante :

pour chaque fichier listé, on applique successivement les filtres dans l'ordre d'apparition dans la commande ; un filtre renvoie un code de retour égal à 0 ou 1 suivant que la propriété est vérifiée ou non.

- Si le code est nul, on passe au deuxième filtre, et ainsi de suite.
- Si tous sont valides, le nom du fichier est fourni en sortie (voir l'action du filtre `-print` plus loin)

III.2.6.

Opérateurs

Un filtre étant un opérateur binaire (son résultat est valide ou invalide), il est possible d'en combiner plusieurs avec les opérateurs NON et OU logique.

Exemples :

`! -filtre` a la validité inverse de `-filtre`

`\( -filtre1 -o -filtre2 \)` est valide si l'un des deux filtres est valide.

(Les parenthèses étant des caractères spéciaux du shell, il faut les protéger avec un ****).

```
find .\( -name '*.c'' -o -name '*.o' \) -mtime +3 -print
```

imprime le nom des fichiers de suffixe `.c` ou `.o` vieux de plus de trois jours.

Commandes

Il existe des filtres particuliers qui consistent à exécuter une commande :

`-print` : affiche le nom du fichier courant.

Ce filtre est toujours valide (son code de retour est toujours nul)

`-exec` : ce filtre spécifie une commande UNIX à exécuter.

Le filtre est valide si le code de retour de la commande UNIX est nul (commande "réussie").

La commande UNIX peut comporter des arguments et doit être suivie d'un blanc et d'un point-virgule pour repérer sa fin. Comme le shell interprète les point-virgules, il faut les protéger avec un ****

Exemple :

```
find /users -exec date \;
```

exécute la commande `date` pour chaque fichier du répertoire contenu dans `/users`

On peut passer à la commande UNIX le nom du fichier en cours, symbolisé par la séquence **{}**

Exemple :

```
find . -exec chmod g+w {} \;
```

donne les droits en écriture pour la classe groupe à tous les fichiers et sous-répertoires du répertoire local (**.**).

Il existe d'autres types de filtres de commande, mais moins utiles.

Agencement des filtres

Comme on l'a vu précédemment, les filtres sont évalués un à un dans l'ordre, donc :

```
find -users -print -name '*.c'
```

affiche tous les noms de fichiers existant sous /users, alors que

III.2.7.

```
find /users -name '*.c' -print
```

affiche uniquement les noms de fichiers de suffixe `.c`

Cette remarque est valable quelque soit le type du filtre utilisé.

Exemple complexe

On utilise la commande UNIX `grep -q` *chaîne* qui renvoie uniquement le code de retour positionné suivant que la *chaîne* a été trouvée ou pas dans le fichier spécifié.

```
find . -exec grep -q chaîne {} \; ! -exec rm {} \; -print
```

Pour chaque fichier trouvé sous le répertoire courant :

- le filtre `-exec grep -q` *chaîne* `{} \;` est évalué ; si *chaîne* est trouvée dans le fichier, le code de retour est nul, et le filtre est valide : on passe au filtre suivant.
- le filtre `-exec rm {} \;` est évalué.

  Si la commande `rm` sur le fichier courant s’est bien passée, le code de retour est nul, et le filtre `-exec` est valide ; donc, le filtre `! -exec` est invalide, et le filtre suivant (`-print`) n’est pas évalué.

  Si la commande `rm` s’est mal passée, -exec est invalide, d’où `!-exec` est valide, et donc -print est évalué.

voir

En conclusion : cette commande supprime tous les fichiers contenant *chaîne* du répertoire courant (ainsi que ses sous-répertoires), et si la suppression est impossible, liste le nom du fichier correspondant.

Options

Voir le manuel(1) pour la liste des filtres et options, qui est très fournie.

III.2.7. grep

But

Rechercher dans un ou plusieurs fichiers une chaîne particulière.

Utilisation

pour un fichier :

`grep` *`ma_chaine mon_fichier`*

Le résultat est un fichier constitué de la suite des lignes dans lesquelles se trouve *`ma_chaine.`*

Pour plusieurs fichiers :

`grep` *`ma_chaine mon_fic1 mon_fic2 mon_fic3 ...`*

Le résultat est un fichier constitué d'une suite de lignes de la forme suivante :

<*nom du fichier*> : <*ligne contenant ma_chaine*>

Quelques options

-c	donne le nombre de lignes contenant `ma_chaine.`
-l	seuls les noms des fichiers contenant `ma_chaine` sont fournis.
-q	rien n'est affiché ; seul le code de retour est positionné suivant que `ma_chaine` a été trouvée ou pas.
-i	pas de distinction entre les majuscules et minuscules pendant la recherche.
-v	seules les lignes ne contenant pas `ma_chaine` sont listées.

III.2.8. head

But

Donne les premières lignes d'un fichier.

```
head mon_fichier
```

donne les 10 premières (par défaut) lignes de `mon_fichier`.

Quelques options

-*N*	indique le nombre de lignes *N* à imprimer.

III.2.9. join

But

En supposant qu'on ait deux fichiers organisés en colonnes (de la même façon qu'on aurait deux tables SQL), `join` permet de faire une jointure entre ces deux fichiers sur une colonne commune.

Utilisation

Les deux fichiers doivent être préalablement triés sur la colonne commune qui servira de jointure.

III.2.9. **Exemple :**

Voici les deux fichiers sur lesquels on veut faire une jointure :

```
$ more j_tab*
::::::::::::::
j_tab1
::::::::::::::
205     Cabours    Paimpol
R4      Dupond     Paris
ferrari Durand     Nice
2CV     Jules      Lourdes
405     Pierre     Lyon
::::::::::::::
j_tab2
::::::::::::::
Cabours      62
Dupond       53
Durand       22
Paul         18
Pierre       35
$
```

La jointure est à faire sur le nom, c’est à dire sur la deuxième colonne de `j_tab1`, et la première de `j_tab2`.

Les fichiers tels qu’ils sont présentés ont été triés :

- sur la deuxième colonne pour `j_tab1`[1]

    `sort -o j_tab1 -b +1 j_tab1`

- sur la première colonne pour `j_tab2`

    `sort -o j_tab2 j_tab2`

(pour le détail, voir la commande `sort` plus loin)

La commande de jointure permettant d’obtenir en sortie toutes les informations des deux fichiers est la suivante :

voir

```
$ join -1 2 -2 1 -o 1.2 2.2 1.3 1.1 j_tab1 j_tab2
Cabours 62 Paimpol 205
Dupond 53 Paris R4
Durand 22 Nice ferrari
Pierre 35 Lyon 405
$
```

1. l’option `-b` permet de ne pas tenir compte des blancs ou tabulations multiples entre les champs; voir le détail avec la commande `sort` page 126

Options utilisées : III.2.10.

`-1 2` indique que la colonne 2 du fichier 1 (dans l’ordre des arguments de la commande `join`) sert de jointure.

`-2 1` indique que la colonne 1 du fichier 2 sert de jointure.

`-o 1.2 2.2 1.3 1.1`

indique dans l’ordre les différentes colonnes à afficher en sortie.

Le symbolisme est le suivant : *f.c*

c est le numéro de colonne du fichier *f*.

Les numéros de colonne démarrent à 1.

Autre option :

`-v` *f* seules les lignes du fichier numéroté *f* qui n’ont pu être appariées sont imprimées.

voir

```
$ join -1 2 -2 1 -v 1 j_tab1 j_tab2
Jules 2CV Lourdes
$ join -1 2 -2 1 -v 2 j_tab1 j_tab2
Paul 18
$
```

III.2.10. nohup

But

Eviter qu’une commande très longue, lancée pendant une session, ne soit tuée lors de la déconnexion.

Utilisation

Il suffit de rajouter “nohup” devant la commande que vous désirez lancer.

nohup *ma_commande* &

Exemple

```
$ nohup id
Sending output to nohup.out
$ nohup "id ;pwd"
Sending output to nohup.out
nohup: id;pwd: No such file or directory
$ more nohup.out
uid=207(logis) gid=20(users)
$
```

III.2.11. Attention, votre commande doit être un fichier exécutable (ce n'est pas possible de lancer une fonction du shell avec `nohup`), et ne doit pas être composée (avec des points virgules par exemple).

Si votre commande doit envoyer des résultats sur la console, ils seront automatiquement déroutés vers un fichier nommé `nohup.out` que vous trouverez dans le répertoire courant si le fichier peut y être mis, ou dans votre répertoire principal sinon.

lancement en arrière plan

Vous utilisez `nohup` généralement lorsque vous vous apprêtez à lancer une commande qui prendra beaucoup de temps avant de se terminer. C'est donc l'occasion de rajouter en fin de ligne le caractère `&` qui lance la commande en arrière plan, et vous permet de reprendre la main de suite.

commandes composées

Pour lancer par `nohup` des commandes composées, il faut passer par l'intermédiaire du shell.

Exemple :

```
$ nohup ksh -c "id ;pwd"
Sending output to nohup.out
$ more nohup.out
uid=207(logis) gid=20(users)
/users/petrus/logis/doc/manuel_shell/exemples
$
```

Il est bien sûr possible de rajouter un & en fin de ligne pour lancer le tout en arrière-plan et reprendre la main de suite.

III.2.11. sed

But

`sed` est la version non intéractive de l'éditeur de lignes `ed`. Il reprend donc la plupart des fonctions de ce dernier. On peut l'utiliser pour modifier de manière automatique le contenu de fichiers.

Cet éditeur proposant un grand nombre de fonctionnalités, on se contentera ici d'en signaler quelques-unes des plus intéressantes :

- la transformation d'une suite de caractères pour un ensemble de lignes d'un fichier
- la suppression d'un ensemble de lignes
- la duplication d'un groupe de lignes à l'intérieur d'un fichier.

III.2.11.

Utilisation

On indique à `sed` une adresse ou un ensemble d'adresses correspondant aux lignes à traiter, puis la commande à exécuter sur chacune des lignes sélectées. Dans le mode normal, les lignes non sélectées sont recopiées telles quelles sur la sortie standard.[1]

Description des adresses

On peut désigner les lignes à traiter :

- par le numéro de la ligne ou le caractère **$** qui symbolise la dernière ligne du fichier.

- par un contenu commun aux différentes lignes :

  /*expression_régulière*/

  (c'est à dire une expression régulière décrivant le contenu entre 2 caractères **/**)

  Si vous voulez utiliser un autre caractère que le **/**, il faut alors écrire :

  \\+*expression_régulière*+

  pour utiliser un caractère **+** comme séparateur.

- par un bloc, c'est à dire une suite de lignes dont la première et la dernière sont définies comme dans les deux cas ci-dessus, et séparées par un caractère virgule **,**

  Si plusieurs blocs du type spécifié sont présents dans le fichier, ils sont tous pris en compte.

Si aucune adresse n'est spécifiée, tout le fichier est traité.

Exemples

128	la ligne 128 du fichier
128,241	les lignes 128 à 241 (comprises)
128,$	toutes les lignes depuis la ligne 128 jusqu'à la fin du fichier
/^[^0-9]*$/[2]	les lignes qui ne contiennent aucun chiffre
/debut/,/fin/	les lignes contenues dans les blocs commençant par une ligne contenant la chaîne "debut" et finissant par une ligne contenant la chaîne "fin".

1. Pour éviter que les lignes non sélectées soient recopiées en sortie, il faut appeler `sed` avec l'option `-n`.
2. la séquence [0-9] désigne un chiffre, donc [^0-9] désigne tout caractère autre qu'un chiffre.
Entre deux **/**, **^** symbolise le début de la ligne, et **$** la fin. Le caractère ***** désigne une chaîne de longueur quelconque constituée uniquement de ce qui précède le ***** dans l'expression régulière.
Donc l'ensemble correspond aux lignes qui ne comprennent que des caractères autre qu'un chiffre entre le début et la fin de ligne.

III.2.11.

Description de quelques commandes

Substitution

s/*expression_régulière*/*chaine_remplacement*/g

s désignation de la commande de substitution

/ le séparateur entre les deux termes qui suivent. Ce n'est pas obligatoirement un / : le séparateur est le caractère qui vient immédiatement après le **s** . Généralement, on utilise un /, mais si l'expression régulière ou la chaîne de remplacement contient un /, il vaut mieux utiliser un autre caractère (dans ce cas, les trois / sont à remplacer)

expression_régulière

une expression régulière qui représente la zone de texte à remplacer. Cette chaîne de recherche est souvent appelée “pattern”.

chaine_remplacement

la chaîne qui remplacera la zone de texte repérée par l'expression régulière

g optionnel. Indique que toutes les occurrences de l'expression régulière rencontrées sur la ligne sont transformées. Si **g** est absent, seule la première est transformée.

Exemple complet

Remplacer dans toutes les lignes qui contiennent des chiffres, chaque chiffre par un 'X'

```
sed '/[0-9]/s+[0-9]+X+g'
```

où on reconnaît :

`/[0-9]/` adresse correspondant aux lignes contenant des chiffres

s+ + + la commande de substitution avec ses séparateurs

`[0-9]` l'expression régulière qui désigne un chiffre

X le caractère qui doit remplacer le chiffre désigné par l'expression régulière

g l'option qui indique de modifier tous les chiffres apparaissant dans une ligne.

Suppression

d commande de suppression de ligne. Les lignes non supprimées sont envoyées sur la sortie standard de la commande `sed`.

Exemple complet

Supprimer la partie du fichier, depuis la ligne contenant la chaîne "debut", jusqu'à la dernière ligne du fichier.

```
sed '/debut/,$d'
```

III.2.11.

où on reconnaît :

/debut/ adresse correspondant à une ligne contenant la chaîne "debut"

$ adresse symbolisant la dernière ligne du fichier.

d la commande de suppression.

Duplication

p commande de duplication de ligne. La ligne dont l'adresse est valide est recopiée sur la sortie standard.

Exemples complets

Dupliquer chaque ligne du fichier, depuis la première ligne jusqu'à la ligne contenant la chaîne "fin".

```
sed '1,/fin/p'
```

où on reconnaît :

1 adresse de la première ligne du fichier.

/fin/ adresse correspondant à une ligne contenant la chaîne "fin"

p la commande de duplication.

Les lignes non reférencées par l'indication des adresses sont quand même envoyées sur la sortie standard, ce qui veut dire que :

- toutes les lignes apparaissent au moins une fois sur la sortie standard
- les lignes dont l'adresse est valide apparaissent deux fois.

Conserver la partie du fichier, depuis la première ligne jusqu'à la ligne contenant la chaîne "fin".

```
sed -n '1,/fin/p'
```

où on reconnaît la même commande que précédemment, à part l'option :

-n qui indique que les lignes dont l'adresse n'est pas sélectionnée ne sont pas recopiées en sortie.

Seules les lignes reférencées par l'indication des adresses sont donc envoyées sur la sortie standard

Utilisation des buffer pour la commande de substitution

Le principe décrit pour une substitution consiste à repérer une chaîne particulière, et à la remplacer par une autre.

Un problème se pose lorsque la chaîne de remplacement doit intégrer une partie de la chaîne à substituer.

Exemple :

dans un fichier, on veut remplacer tout nombre négatif par le même nombre, mais rendu positif (par suppression du "tiret" qui le précède).

III.2.11. La chaîne à rechercher est un "tiret" suivi d'un chiffre.

La chaîne de remplacement est le chiffre trouvé dans la chaîne à rechercher (sans le tiret).

Donc, il faut pouvoir, lors de la recherche, mettre en mémoire une partie de la chaîne à substituer.

`sed` permet de placer dans des buffers des parties de la chaîne à substituer, pour les intégrer à la chaîne de remplacement.

buffer principal

Pendant une substitution, toute chaîne qui satisfait à l'expression régulière de recherche est placée dans le buffer principal.

Le contenu du buffer peut être placé dans la chaîne de remplacement en utilisant le caractère **&**.

Exemple :

```
$ more j_tab2
Cabours 62
Dupond 53
Durand 22
Paul 18
Pierre 35
$ sed 's/[0-9][0-9]*/& &/' j_tab2
Cabours 62 62
Dupond 53 53
Durand 22 22
Paul 18 18
Pierre 35 35
$
```

Ici, l'expression régulière correspond à une suite de caractères composée :

- d'un chiffre

- suivi d'un nombre positif ou nul d'autres chiffres

c'est à dire : au moins un chiffre.

D'autre part, la chaîne de remplacement est égale à deux fois le contenu du buffer, séparé par un <espace>.

Donc dans le fichier de sortie, chaque suite de chiffre est remplacée par deux fois elle même, plus un <espace> entre les deux.

Buffers secondaires

Pour récupérer une partie seulement de la chaîne de recherche, on utilise un buffer secondaire. On indique à l'intérieur de l'expression régulière la partie de la chaîne à mettre dans le buffer secondaire en l'encadrant entre des :

`\(      \)`

Dans la chaîne de remplacement, le contenu du buffer est désigné par :

`\1`

La présence de **\(\)** ne modifie en rien la signification de l'expression régulière : la seule action est donc de placer dans le buffer la partie de la ligne correspondant à la partie d'expression régulière encadrée par les **\(\)**. III.2.11.

Exemple du changement de signe négatif en positif en utilisant un buffer secondaire :

```
$ more chg_sign
ESSAI-8 -11
jean-francois +10
valeur -moins -5
$ sed 's/-\([0-9][0-9]*\)/\1/g' chg_sign
ESSAI8 11
jean-francois +10
valeur -moins 5
$
```

La chaîne recherchée est :

	-	: un tiret
suivi par	`[0-9]`	: un caractère entre 0 et 9 (un chiffre donc)
suivi par	`[0-9]*`	: un chiffre répété 0 ou n fois

Lorsqu'elle est localisée, la séquence `[0-9][0-9]*` (c'est à dire, une série d'au moins un chiffre) est placée dans le buffer, puisque cette séquence est encadrée par des `\( \)`.

La chaîne de remplacement est le buffer.

La chaîne de recherche est un tiret suivi de chiffres ; la chaîne de remplacement ne contient plus que les chiffres : on supprime le tiret devant les termes constitués d'au moins un chiffre.

<u>On remarque dans le résultat :</u>

- que les chaînes `jean-francois` et `-moins` n'ont pas été transformées, ce qui est conforme aux "spécifications" ;
- par contre, que la chaîne `ESSAI-8` a été modifiée, ce qui n'est peut-être pas tout à fait l'objectif de départ.[1]

Buffers secondaires multiples

Il est possible de définir plusieurs buffers secondaires : il suffit de faire apparaître plusieurs séquences **\(\)** dans la chaîne de recherche. La première séquence est notée **\1** dans la chaîne de remplacement, la seconde **\2**, etc...

1. En effet, un nombre tel qu'on l'imagine est une suite de chiffres éventuellement précédée d'un tiret, mais isolée entre des séparateurs.
Donc, dans la chaîne de recherche, il aurait fallu ajouter un caractère <espace> ou <tabulation> avant et après la séquence -[0-9][0-9]*.
Mais dans ce cas, on ne tient pas compte des "nombres" collés en début ou en fin de ligne (pas de séparateur)...
En fait, la recherche consisterait à trouver un champ constitué uniquement de chiffres (et précédés éventuellement du caractère tiret). Pour cela, il faut savoir que la commande `awk` est mieux adaptée (l'accès aux différents champs est immédiat... : voir la fonction `gsub` de `awk`)

III.2.11. Exemple d’échange de position pour un couple de chiffres :

```
$ more j_tab2
Cabours 62
Dupond 53
Durand 22
Paul 18
Pierre 35
$ sed ’s/\([0-9]\)\([0-9]\)/\2\1/’ j_tab2
Cabours 26
Dupond 35
Durand 22
Paul 81
Pierre 53
$
```

Les deux chiffres consécutifs sont placés chacun dans un buffer secondaire, et on remplace le couple par lui même, mais dans l’ordre inverse (...**\2\1**..)

<u>Remarques générales sur l’utilisation des substitutions avec `sed`</u>

- Les chaînes de recherche de `sed` sont utilisables en priorité sur des suites de caractères : si vous devez isoler des *champs*, choisissez plutôt la commande `awk` qui est mieux adaptée.
- Dans tous les exemples, la chaîne de commande passée à `sed` est encadrée par des quotes simples. Ceci n’est pas indispensable, mais attention : beaucoup de caractères spéciaux pour `sed` (***,$,\\,[],...**) le sont aussi pour le shell, qui risque de les interpréter avant de passer le résultat au `sed` (dans le cas où la chaîne n’est pas protégée par des quotes).

  Ainsi, on peut remplacer les quotes simples par des quotes doubles si la chaîne de commande ne contient pas de **$** ou de **"**, et ne pas en mettre si la chaîne de commande ne contient aucun des caractères spéciaux, ni de séparateurs <espace> ou <tabulation>.

A l’inverse, vous pouvez construire la chaîne de commande du `sed` avec le contenu d’une variable. Exemple :

```
$ CONSOLE=`tty`
$ echo "le nom de ma console est <cons>"
le nom de ma console est <cons>
$ echo $CONSOLE
/dev/pty/ttyt1
$ echo "le nom de ma console est <cons>" |
> sed "s+<cons>+$CONSOLE+"
le nom de ma console est /dev/pty/ttyt1
$
```

Au moment de l'évaluation de la ligne contenant le sed, le shell remplace dans la chaîne "s+<cons>+$CONSOLE+" la variable $CONSOLE par son contenu, puisque celle-ci est encadrée par des ". III.2.12.

Elle devient donc : s+<cons>+/dev/pty/ttyt1+, suite à quoi, le shell lance la commande sed s+<cons>+/dev/pty/ttyt1+.

La présence de **/** dans la chaîne de remplacement impose l'utilisation de **+** comme caractère de séparation.

Une chaîne de commande passée à sed est une chaîne comme une autre : vous pouvez donc la composer à partir de variables, de résultats d'une autre commande, de chaînes fixes, le tout juxtaposé.

Mais, comme on l'a déjà signalé dans un chapitre précédent[1], les caractères spéciaux du shell, ou les <espace>, qui ne sont pas protégés, seront tout d'abord interprétés par le shell et provoqueront donc souvent une erreur de syntaxe dans le sed.

C'est pourquoi le mieux dans la majorité des cas est de d'encadrer les commandes de sed avec des simple-quotes.

III.2.12. sort

But

Trier un fichier.

Principe

Le mode naturel du tri consiste à comparer le premier caractère de chaque ligne et de les ordonner en conséquence. Pour toutes les lignes dont le premier caractère est identique, on recommence la comparaison sur le deuxième caractère, et ainsi de suite.

Dans le cas du tri numérique, un traitement de logique interne tabule à droite les nombres au cas où ceux-ci seraient de longueur variable.

fichier d'entrée

Il est possible de spécifier le fichier à trier de plusieurs façons :

sort *mon_entree*

ou bien sort <*mon__entree*

ou encore cat *mon__entree* | sort

1. Voir l'interprétation d'une ligne shell : page 34

III.2.12. **fichier de sortie**

sort <*mon_entree* >*ma_sortie*

ou bien sort *mon_entree* -o *ma_sortie*

Comme `sort` utilise des fichiers temporaires pour effectuer ses manipulations, il est possible de donner comme fichier de sortie celui d'entrée.

sort *mon_fichier* -o *mon_fichier*

Utilisation simple

Pour les exemples qui suivront, le fichier d'entrée sera toujours le même :

```
$ more sort.input
2 45
1 126
1  45
 2 126
1 7
$
```

(Attention aux espaces, qui sont significatifs dans la plupart des cas).

Exemple trivial :

```
$ sort sort.input
 2 126
1  45
1 126
1 7
2 45
$
```

Par défaut, le tri est alphanumérique (dans l'ordre des caractères ascii), et commence à partir du premier caractère à gauche pour chaque ligne.

Ici, la première ligne commençant par un <espace>, elle apparait donc en première position[1].

Pour les trois lignes débutant par un 1, le deuxième caractère n'est pas suffisant pour assurer le tri : c'est un <espace>. Donc, ce sera le troisième qui permettra de départager : un <espace>, un 1, un 7.

1. l'<espace> est un caractère ascii de rang inférieur à tous les caractères alphabétiques et numériques. La liste est consultable dans le fichier `/usr/pub/ascii` sur votre machine.

tri numérique III.2.12.

Le tri ne tient pas compte des <espace> ou <tabulation> qui précèdent le terme numérique considéré.

```
$ sort -n sort.input
1  45
1 126
1 7
 2 126
2 45
$
```

On constate bien que les lignes qui commencent par un 1 précèdent celles qui commencent par un 2 (!)

Par contre, il apparait que le tri numérique ne s'est pas effectué sur la deuxième colonne : le tri alphanumérique a été repris à compter du premier <espace> ou <tabulation> qui suit le premier champ.

clé

Le tri par défaut consiste à trier à partir du premier caractère de chaque ligne, et la ligne complète est utilisée si les premiers caractères ne suffisent pas.

On peut limiter le tri à une partie de ligne. Cette partie qui est alors appelée la *clé*.

Une clé est repérée dans la ligne par une position de début (inclue) et une position de fin (non inclue).

Juste avant le tri, la ligne est découpée de manière logique en champs, numérotés à partir de zéro, et séparés par des <espace> ou <tabulation>[1].

Un champ commence en début de ligne, ou à partir de la position qui suit l' <espace> ou la <tabulation> définissant la fin du champ précédent. (ce qui veut dire que si deux champs sont séparés par 3 <espace>, alors le deuxième champ comprendra 2 <espace> comme premiers caractères). Le dernier caractère du champ précède toujours un <espace>, une <tabulation>, ou la fin de ligne.

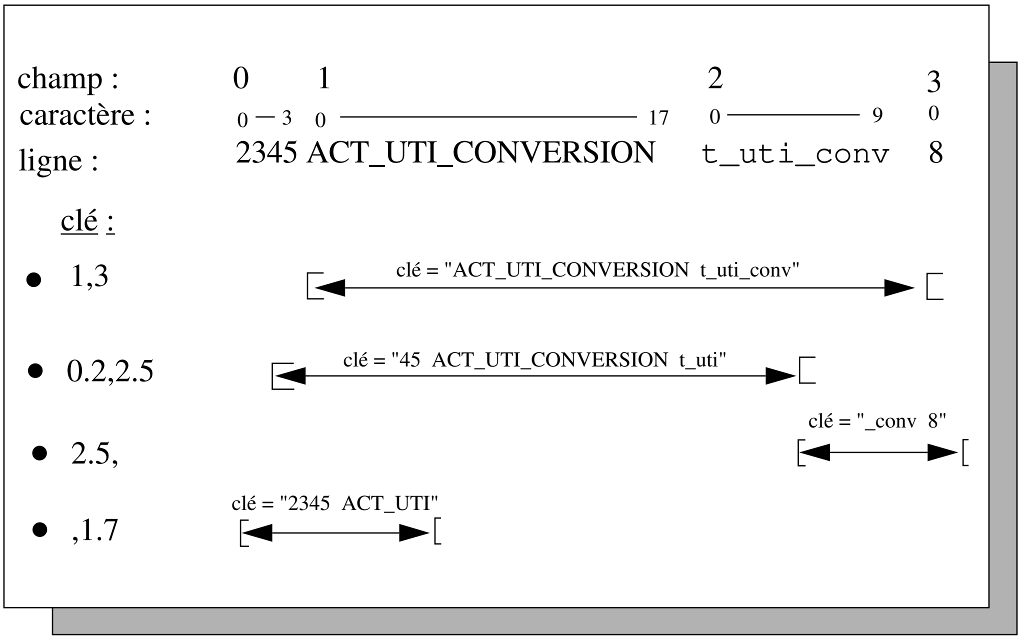
Dans chaque champ, les caractères sont numérotés, à partir de zéro eux aussi.

Une clé se définit de la manière suivante :

chp_debut.car_debut,chp_fin.car_fin

1. Il est possible de définir d'autres caractères de séparation que <espace> et <tabulation>. Voir l'option `-t`

III.2.12. Si on omet `car_debut` ou `car_fin`, tout le champ est considéré. Si on omet respectivement le terme avant ou après la virgule, le début ou la fin de la ligne est considérée. A l'intérieur de la clé, les <espace> ou <tabulation> sont pris en compte.



tri restreint

Pour utiliser une clé dans le tri, on spécifie l'option -k[1] :

voir voir

```
sort -kcle1 -kcle2 -kcle3 ...
```

On peut donner plusieurs clés, au cas où le fichier contiendrait des lignes pour lesquelles la première clé (voire la seconde, etc...) serait identique.

Rien n'empêche de trier d'abord sur une clé basée sur la fin de la ligne, puis, pour affiner, sur une clé basée sur le début de la ligne.

Exemple simple :

voir

voir

tri sur le champ No 1

```
$ sort -k1, sort.input
1  45
 2 126
1 126
2 45
1 7
$
```

champ No 1 une fois cadré
(1 séparateur oté)

```
 45
126
126
45
7
```

1. Par compatibilité avec les versions antérieures, la clé peut aussi être spécifiée par :
`sort +chp_debut.car_debut -chp_fin.car_fin`

Remarquez bien que le tri étant ici alphanumérique (par défaut), il est normal que le champ commençant par un 1 (126) précède celui qui commence par un 4 (45), même si le premier nombre est supérieur au second. III.2.12.

Le champ numéroté 1 étant le dernier de la ligne, il n'est pas nécessaire d'indiquer où se termine la clé ; mais on aurait pu aussi bien taper :

voir voir

```
sort -k1,2 sort.input
```

<u>Suppression des séparateurs multiples</u>

Pour s'affranchir du problème posé par des <espace> ou <tabulation> répétés entre deux clés et qui perturbent le tri, on utilise l'option `b`.

Pour cela on rajoute `-b` comme option du `sort`, (dans ce cas, la clé qui suit sur la ligne est concernée), ou ce qui est mieux, dans la spécification de la clé :

voir

voir

tri sur le champ No 1
en supprimant les
séparateurs multiples

```
$ sort -kb1, sort.input
 2 126
1 126
1  45
2 45
1 7
$
```

champ No 1 une fois cadré
(tous les séparateurs otés)

```
126
126
45
45
7
```

On aurait pu aussi taper :

```
sort -b -k1, sort.input
```

Attention :

- cette option n'est valide que si une clé au moins est définie.

  (`sort -b sort.input` n'a pas d'effet)

- à l'intérieur de la clé, les séparateurs ne sont pas ignorés.

<u>tri multiple</u>

Exemple de tri sur le champ No 1, puis sur le champ No 0 :

voir

voir

```
$ sort -b -k1,2 -k0,1 sort.input
1 126
 2 126
1  45
2 45
1 7
$
```

III.2.12. **<u>options du tri</u>**

Les options (comme `b`) peuvent être spécifiées par :

`-`*<u>`opt`</u>* comme une option de la commande `sort`

sa validité porte alors sur la clé qui suit sur la ligne de commande. Donc `sort -`*<u>`opt`</u>* *`clé`* n'a pas la même signification que `sort` *`clé`* `-`*<u>`opt`</u>*

Attention, le résultat n'est pas toujours garanti : suivant les options et manipulations, il arrive que l'option porte sur une clé, ou parfois toutes celles qui suivent.

`-k`*<u>`opt`</u>**clé* sa validité ne porte que sur la *clé.*

A utiliser de préférence.

Voici les plus intéressantes :

`b` déjà vue. supprime les séparateurs multiples en début du champ si elle s'applique sur une clé. Attention, à l'intérieur de la clé, les séparateurs ne sont pas ignorés.

`n` tri numérique. Le champ concerné est tabulé à droite.

`r` renverse le sens du tri de la clé spécifiée, ou de la totalité du tri si aucune clé n'est déclarée.

`f` pour la comparaison, toutes les lettres sont passées en majuscules.

<u>Exemple complexe</u>

On veut trier le fichier précédent :

- sans tenir compte des séparateurs multiples
- d'abord en numérique et inversé sur le champ numéroté 1
- ensuite en alphanumérique sur le champ numéroté 0 :

voir

voir

```
$ sort -krn1, -k0,1 sort.input
 2 126
1 126
1  45
2 45
1 7
$
```

On vérifie bien que les valeurs numériques du champ 1 décroissent et que dans le champ 0 des deux premières lignes (où il y a ambiguïté sur le champ 1), l'<espace> précède le caractère 1.

III.2.13.

autres options de la commande sort

`-u` après le tri, les lignes identiques dupliquées (doublons) sont supprimées : une seule occurrence est conservée.

`-t`*x* le caractère *x* est le séparateur, à la place de l'<espace> et de la <tabulation> pris par défaut.

`-c` teste si le fichier d'entrée est conforme au tri spécifié.

renvoie 0 si le fichier est déjà correctement trié, ou 1 avec un message d'erreur sinon.

III.2.13. strings

But

Lister les chaînes de caractères (de type *constante*) qui sont inclues dans un fichier binaire, ou un fichier core[1].

Utilisation

`strings` *mon_fichier* | `more`

le pipe suivi du `more` n'est absolument pas indispensable, mais bien utile, car la liste est souvent longue.

Option

-*N* indique le nombre de caractères *N* à partir de laquelle une chaîne est extraite. Par défaut cette valeur vaut 4, et seules les chaînes d'au moins 4 caractères sont indiquées.

Exemples de chaînes extraites

- binaire ou core issu d'un programme source en C :

```
...
start %20s ...
done after usr=%5u sys=%5u
exit %s
newSubclass %s
...
```

=> les chaînes contenant des %s , %d, %c, etc ... sont typiques du langage C (format d'un `printf`). L'exécutable contient certainement un module provenant du langage C.

- binaire ou core issu d'un programme source en ADA :

1. voir la signification du terme `core` au paragraphe sur les messages d'erreur, page 209, à : `core dumped`

III.2.13.

```
...
to a non Ada scope.
-- Program terminated by an exception propagated
out of the main subprogram.
------------- Stack trace of the propagation in the
main stack --------------
----------------------- skipped scopes: -----------
-----------
CONSTRAINT_ERROR -- Exception raised:

SUBPROGRAM NUMERIC_ERROR STORAGE_ERROR
TASKING_ERROR
PROGRAM_ERROR
 -- End of propagation: Program aborted.
...
```

=> là, pas de doute, c'est écrit noir sur blanc

- le binaire ou un core du KornShell (`ksh`) :

...
rksh
[[!
*macs
$HOME
$PWD
$OLDPWD
...

=> les termes rksh et $OLDPWD sont caractéristiques du ksh. Si votre fichier est un core, il correspond probablement à un programme shell "planté".

- le binaire ou un core du BourneShell (`sh`) :

```
...
sh internal 1K buffer overflow
...
```

=> c'est écrit noir sur blanc.

- le binaire ou un core du Csh :

```
...
```

III.2.14.

```
Not in while/foreach
then
default
then/endif not found
...
```

=> les termes `foreach` et `endif` sont caractéristiques du Csh.

III.2.14. tail

But

Donne les dernières lignes d’un fichier.

Utilisation

`tail` *mon_fichier*

donne les 10 dernières (par défaut) lignes de *mon_fichier*.

Quelques options

-n *N* indique le nombre de lignes *N* à imprimer.

-f met la commande en mode “suivi”.
la commande affiche les lignes au fur et à mesure que le fichier se rempli.

III.2.15. uniq

But

Supprime les occurrences multiples de lignes consécutives dans un fichier trié.

Utilisation

`uniq` *mon_fichier_trié*

conserve une seule occurrence des différentes lignes de *mon_fichier_trié*.

Quelques options

-u seules les lignes qui ne sont pas répétées sont imprimées.

-d seule une copie des lignes répétées est imprimée.

Par défaut, la commande utilise ces deux options réunies.

-*n* les *n* premiers champs sont ignorés pendant la comparaison.

III.2.15. Exemple[1] :

```
$ sort +2 tst_sort1
4 1 1 2
8 0 1 2
4 1 1 3
$ sort +2 tst_sort1 | uniq -2
4 1 1 2
4 1 1 3
$ sort +2 tst_sort1 | uniq -u -2
4 1 1 3
$ sort +2 tst_sort1 | uniq -d -2
4 1 1 2
$ sort tst_sort1 | rev | uniq -2 | rev
4 1 1 2
8 0 1 2
$
```

voir

L'exemple ci-dessus consiste à manipuler un fichier contenant trois lignes de chiffres.

Première manipulation

On veut conserver une seule occurrence des lignes consécutives dont les deux derniers champs sont égaux (comparaison ligne à ligne) ; les autres lignes sont conservées :

- on trie à partir de l'avant dernier champ (`sort +2`), et on compare les lignes en ignorant les deux premiers champs (`uniq -2`)

Deuxième manipulation

On veut éliminer les lignes consécutives ayant les deux derniers champs égaux (comparaison ligne à ligne) :

- la manipulation est identique à la précédente, à part l'option `-u` spécifiée à la commande `uniq`.

Troisième manipulation

Identique à la première manipulation, mais les autres lignes ne sont pas affichées

- on spécifie l'option -d à la commande `uniq`.

Quatrième manipulation

Identique à la première manipulation, mais la comparaison doit porter sur les deux premiers champs :

voir

1. La syntaxe utilisée ici pour la commande `sort` est l'ancienne version, supportée encore aujourd'hui par compatibilité. La syntaxe correcte serait :

```
sort -k2, tst_sort1
```

pour trier depuis le champ numéroté 2 jusqu'à la fin de ligne.

- comme la commande `uniq` ne permet pas ce type de comparaison (ignorer N derniers champs), on inverse les caractères sur chaque ligne (symétrie par rapport au centre, les N derniers champs deviennent les N premiers) avec la commande `rev`, puis on applique `uniq`, puis on inverse à nouveau pour retrouver la ligne dans l'état initial.

Remarques :

- il n'est pas possible en utilisant `uniq` de comparer des lignes consécutives en ignorant des champs à la fois en début et en fin de ligne.
- si le nombre ou la longueur des champs varient de ligne en ligne, l'exemple qui utilise la commande `rev` n'est plus valable.

III.2.16. xargs

But

Lancer autant de fois une commande qu'il existe de paquets d'arguments pour cette commande dans un fichier donné.

Utilisation

Les paramètres de `xargs` sont:

- la commande à lancer;
- le fichier contenant les paquets d'arguments pour cette commande. (`xargs` peut aussi travailler avec l'entrée standard)

La commande à lancer doit être un binaire exécutable ou un shell-script (pas d'alias, ou de commande composée). Si elle n'est pas spécifiée, alors `echo` est exécuté par défaut. On peut indiquer en plus de la commande un ou plusieurs paramètres.

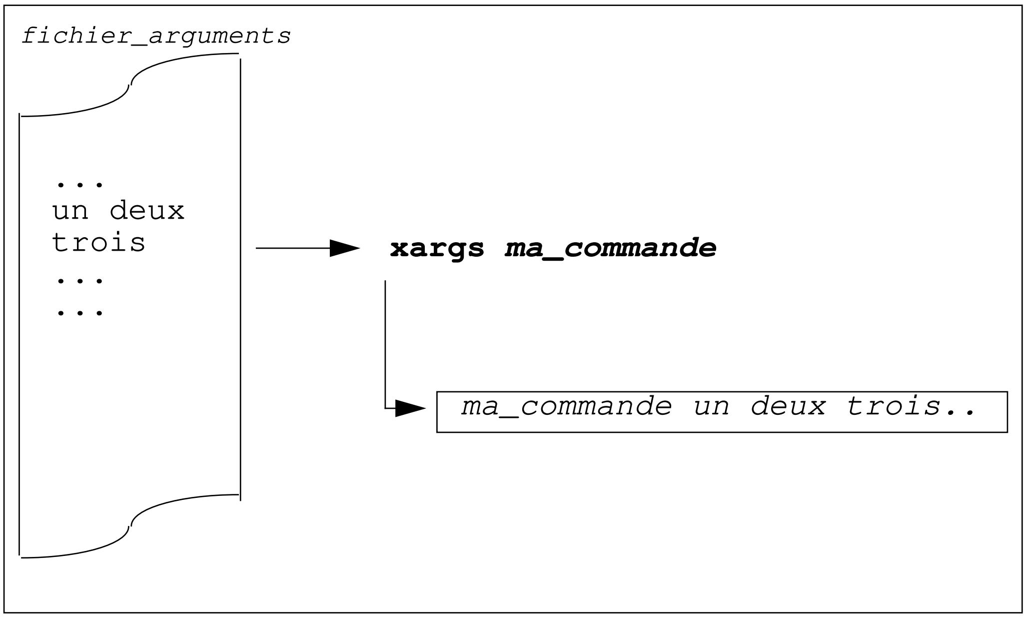
Les données du fichier (ou de l'entrée standard) sont des champs séparés par des <*espace*>, <*tabulation*>, ou des <*retour à la ligne*>. Il est possible de placer un de ces séparateurs dans un champ en encadrant le tout avec des quotes simples ou doubles, ou en le faisant précéder du caractère **** .

Un paquet d'arguments est constitué d'une suite de champs trouvés dans le fichier, séparés par des <*espace*>. Le nombre de champs successifs pris dans le fichier pour faire un paquet dépend de l'option utilisée; en standard (c'est à dire sans option), le paquet est constitué de tous les champs du fichier d'entrée.

Si on prend :

```
cat fichier_arguments | xargs ma_commande
```

III.2.16. Le fonctionnement standard consiste à prendre *ma_commande*, à rajouter derrière tous les champs du fichier *fichier_arguments*, et à exécuter le tout.



Si *fichier_arguments* contient 25 champs, *ma_commande* sera lancée avec ces 25 champs en paramètres.

Exemple :

```
$ cat tst_xargs
un deux trois
quatre
cinq six
$ xargs echo AA <tst_xargs
AA un deux trois quatre cinq six
$
```

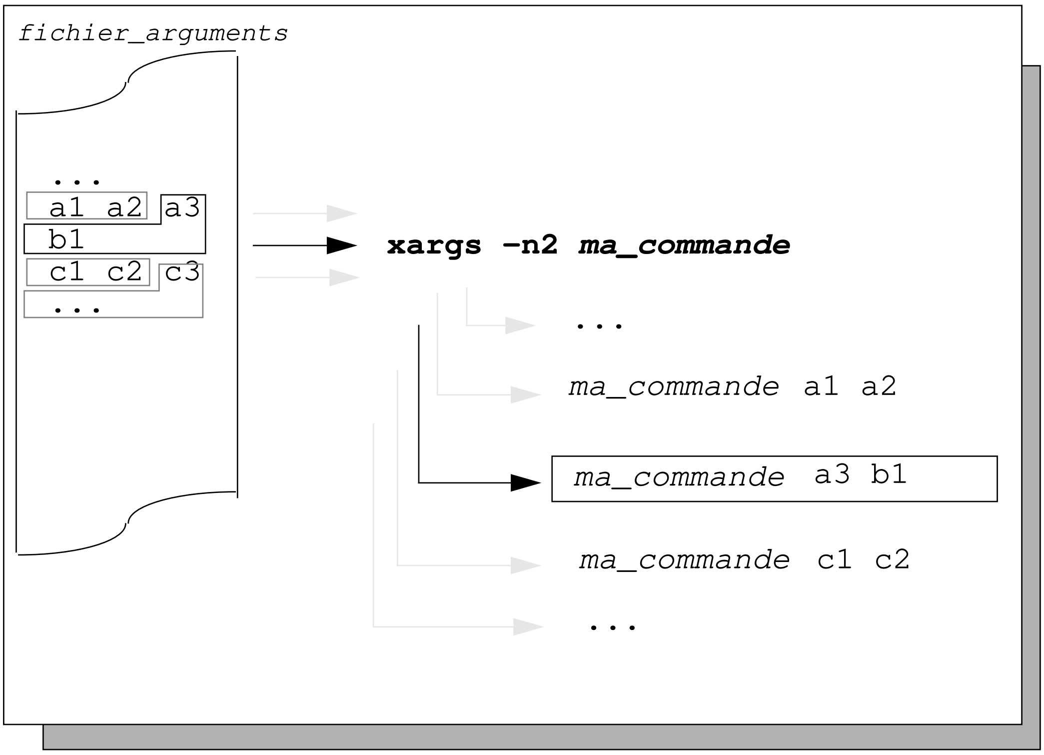
Quelques options III.2.16.

-n*N* Au lieu de fabriquer un seul paquet avec tous les champs du fichier, `xargs` constitue des paquets de *N* champs (sauf le dernier qui contient ce qui reste) et lance la commande autant de fois qu’il y a de paquets.

Si on prend :

`cat` *`fichier_arguments`* `|` `xargs` *`-n2 ma_commande`*

`xargs` transforme *`fichier_arguments`* en une suite de paquets de 2 champs, et applique *`ma_commande`* sur chacun de ces paquets.



Exemple:

```
$ cat tst_xargs
un deux trois
quatre
cinq six
$ xargs -n4 echo AA <tst_xargs
AA un deux trois quatre
AA cinq six
$
```

III.2.16.

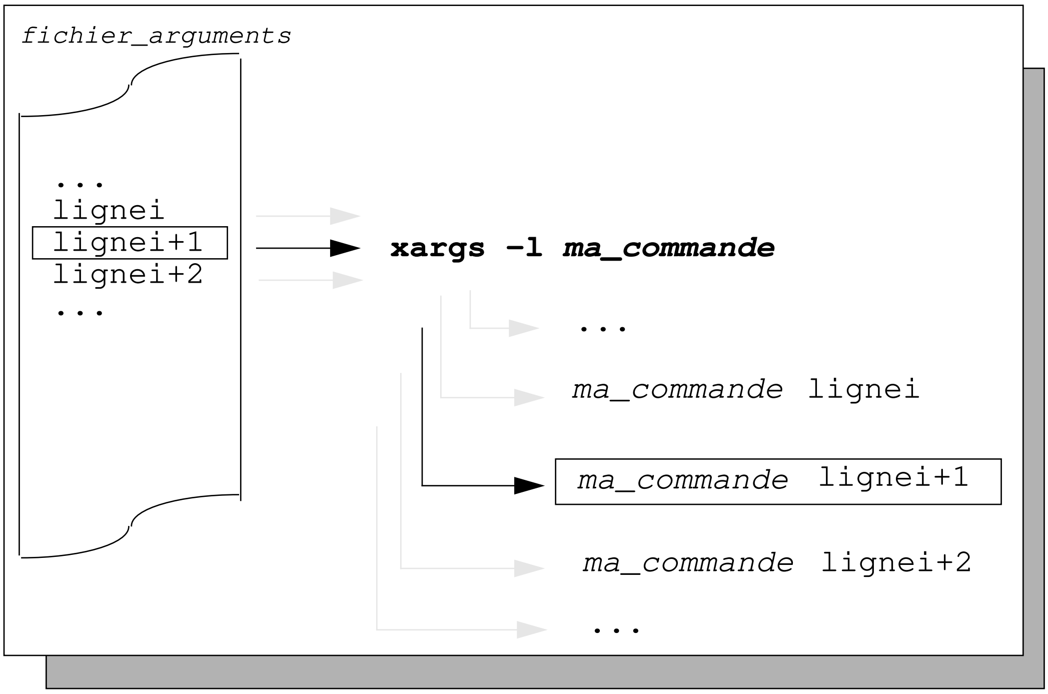
`-l`*N* Au lieu de fabriquer un seul paquet avec tous les champs du fichier, `xargs` constitue un paquet par ensemble de *N* lignes, chaque paquet contenant l’ensemble des champs des *N* lignes correspondantes.

Si *N* est omis, la valeur 1 est prise par défaut.

Si on prend :

`cat` *`fichier_arguments`* `| xargs -l` *`ma_commande`*

Pour chaque ligne du fichier d’entrée *`fichier_arguments`*, le fonctionnement consiste à prendre *`ma_commande`*, à rajouter au bout la ligne courante, et à exécuter le tout.



Exemple:

```
$ cat tst_xargs
un deux trois
quatre
cinq six
$ xargs -l echo AA <tst_xargs
AA un deux trois
AA quatre
AA cinq six
$
```

III.2.16.

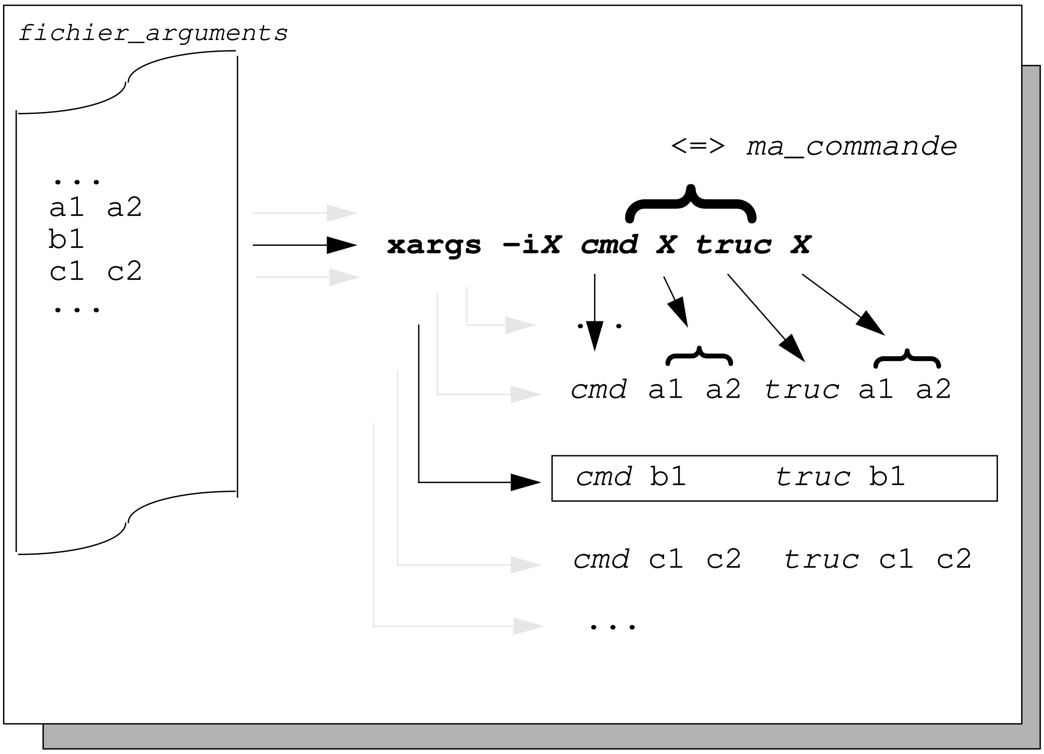
`-i`*`chaine`* Au lieu de rajouter le paquet de champs en bout de *ma_commande*, il est inséré en lieu et place de la chaîne *chaine* dans m*a_commande*. Si *chaine* est omise, la chaîne **{}** est prise par défaut.

Le paquet est constitué de l'ensemble des champs de la ligne courante.

Si on prend :

`cat` *`fichier_arguments`* `|` `xargs -i`*`X cmd X truc X`*

Dans le texte de la commande fournie en argument à xargs, chaque lettre *X* est remplacée par le contenu de la ligne courante de *fichier_arguments*.



Exemple:

```
$ cat tst_xargs
un deux trois
quatre
cinq six
$ xargs -ipaquet echo paquet AA paquet <tst_xargs
un deux trois AA un deux trois
quatre AA quatre
cinq six AA cinq six
$
```

III.2.16. **Remarque :**

Les options -i et -n*N* ne sont pas compatibles :

```
$ echo 'tata tutu' | xargs -i -n1 echo "{} {} {} |"
{} {} {} | tata
{} {} {} | tutu
$ echo 'tata tutu' | xargs -n1 -i echo "{} {} {} |"
tata tutu tata tutu tata tutu |
$
```

alors que le résultat attendu consiste en deux lignes, la première contenant trois “tata”, et la seconde contenant trois “tutu”.

La solution consiste à opérer en deux temps :

echo ’tata tutu’ | xargs -n1 | xargs -i echo “{} {} {} |”

```
$ echo 'tata tutu' | xargs -n1 |
> xargs -i echo "{} {} {} |"
tata tata tata |
tutu tutu tutu |
$
```

- le premier `xargs` applique la commande `echo` (par défaut) champ après champ (option `-n1`) sur le fichier d’entrée (l’entrée standard)
- le deuxième applique un `echo` avec comme arguments trois fois chaque ligne d’entrée.

En bref :

- Vous avez 200 fichiers à renommer: il faut leur rajouter le suffixe `.ext` . La liste des noms de ces fichiers se trouve dans le fichier `liste`, un nom par ligne:

  `xargs -ific mv fic fic.ext <liste`
- Idem, mais les ancien et nouveau noms sont placés par couple, un couple sur chaque ligne du fichier `liste`:

  `xargs -l mv <liste`
- Vous voulez reformater un fichier, en mettant les champs sur trois colonnes:

  `xargs -n3 <fichier`
- Vous voulez reformater un fichier, en joignant les lignes 2 à 2:

  `xargs -l2 <fichier`

III.2.16.

IV. Cas types

IV.1. Cas types dans un programme shell

IV.1.1.

IV.1.1. Comment affecter une variable avec le résultat d'une commande?

1/ le résultat de la commande doit se trouver sur la sortie standard :

si ce n'est pas le cas, rediriger la sortie avec une redirection vers la sortie standard (descripteur 1)

exemple : vous voulez récupérer la sortie d'erreur de la commande `cmd`, dans ce cas la redirection à prévoir est : `cmd 2>&1`

2.1/ utilisez de préférence l'opérateur $( )

exemple : `toto=$(cmd)`

ou, si la sortie de la commande devait être redirigée :

toto=$(cmd 2>&1)

(dans le cas ci-dessus, vous récupérez le résultat de la sortie d'erreur de la commande `cmd`)

Sinon, on peut aussi utiliser le caractère **‘** (anti-quote) :

toto=‘cmd‘

2.2/ au cas où il y aurait plusieurs variables à récupérer sur la ligne de sortie de la commande

utilisez la commande `read`

exemple : `cmd | read var1 var2 var3`

(dans le cas ci-dessus, le résultat de la commande cmd est une ligne, et on récupère le premier champ du résultat de `cmd` dans la variable `var1`, le 2ème dans `var2` et tout ce qui reste dans `var3`)

Exemple : on veut récupérer par la commande `wc -l` le nombre de lignes du fichier toto1.c

```
$ wc -l toto1.c
 2 toto1.c
$ nbl1=$(wc -l toto1.c)
$ echo $nbl1
2 toto1.c
$ wc -l toto1.c | read nbl2 fichier
$ echo $fichier
toto1.c
$ echo $nbl2
2
$
```

informations :

voir guide : les redirections

l'opérateur $() ou anti-quote

les variables

IV.1.2.

IV.1.2. Comment lire un fichier ligne à ligne?

1/ <u>le fichier est constitué par la sortie standard d'une commande `cmd`</u>

Squelette :

```
cmd |
{
   while read ligne
   do
       VOTRE_TRAITEMENT
   done
}
```

Remarque :

Si vous connaissez la structure de votre fichier, vous pouvez récupérer les différents champs directement dans plusieurs variables :

remplacez `read ligne` par `read chp1 chp2 chp3` ...

2/ <u>le fichier est un fichier déjà existant</u> (par ex : toto)

2.1 : Choisir un numéro de descripteur inutilisé et supérieur à 2

par ex : 3

2.2. : Squelette

```
exec 3<toto
while read -u3 ligne
do
      VOTRE_TRAITEMENT
done
```

informations :

voir guide : les redirections

la manipulation des fichiers

les variables

les macro-commandes { } et ( )

la commande `while`

Manuel UNIX : `ksh(1)` (commande interne `read`)

IV.1.3. Comment mettre une trace d'exécution?

1/ à l'écran

rajouter juste avant la zone à tracer :

IV.1.4.

```
set -x
```

et éventuellement après la zone, pour stopper la trace :

```
set +x
```

2/ dans un fichier

idem précédent, mais rajouter avant la commande `set -x` :

```
exec 2>fichier_trace
```

informations :

voir guide : les redirections
la manipulation des fichiers
la commande shell `set`

IV.1.4. Comment tester si un fichier existe? s'il est vide?

1/ le fichier *toto* existe-t-il?

squelette :

```
if [ -a toto ]
then      # fichier existant
          votre_traitement1
else      # fichier inexistant
          votre_traitement2
fi
```

2/ le fichier *toto* est-il vide?

squelette :

```
if [ ! -s toto ]
then      # fichier vide
          votre_traitement1
else      # fichier non vide
          votre_traitement2
fi
```

Il existe une bonne vingtaine de tests différents sur les fichiers

informations :

voir guide : les expressions conditionnelles
la commande `if`

Manuel(1) : `ksh(1)` expressions conditionnelles,
liste des tests possibles

IV.1.5.

IV.1.5. Comment exécuter une commande sur plusieurs fichiers de mon répertoire?

1/ uniquement au niveau du répertoire considéré :

Exemples :

- tous les fichiers du répertoire courant, sans exception :

`ls -1 | xargs` *`commande`*

- les fichiers de suffixe *`.suff`* du sous-répertoire *`tutu`*

`ls -1` *`tutu/*.suff`* `| xargs` *`commande`*

Remarque :

- chez les utilisateurs, la commande `ls` est souvent aliasée pour avoir un listing personnalisé : c'est pour être sur d'avoir un nom de fichier par ligne qu'on utilise l'option `-1`.

2/ sur le répertoire considéré et tous les sous-répertoires

On utilise la commande `find.`

Exemple : lancer sur tous les fichiers du répertoire courant et de ses sous-répertoires la commande *`commande :`*

```
find . -exec commande {} \ ; -print
```

Pour chaque *`fichier`* trouvé par la commande `find`, l'action réalisée est :

```
commande fichier
```

3 / même chose qu'en 2, mais uniquement si le nom du fichier comporte le suffixe *.suff*

```
find . -name '*.suff' -exec commande {} \ ; -print
```

informations :

voir guide : la commande UNIX `xargs`
la commande UNIX `find`
les caractères spéciaux et expressions génériques

Manuel UNIX : `xargs(1)`
`find(1)`

IV.1.6. Comment additionner ou multiplier deux entiers?

1/ En typant la variable qui reçoit le résultat

Exemple :

```
typeset -i toto
titi=2
toto=215+titi*312-4+2
```

2/ En utilisant l'opérateur (( ))

```
titi=2
```

```
(( toto = 215 + titi * 312 - 4 + 2 ))
```

3/ en utilisant l'opérateur $(( ))

```
titi=2
toto=$(( 215 + titi * 312 - 4 + 2 ))
```

informations :

voir guide : les variables
la commande `typeset`
les expressions arithmétiques

IV.1.7. Comment transformer le contenu d'une variable en majuscules? en minuscules?

pour transformer le contenu de la variable *toto* en majuscules :

```
typeset -u toto
```

en minuscules :

```
typeset -l toto
```

informations :

voir guide : les variables
la commande `typeset`

IV.1.8. Comment contraindre une variable à une longueur donnée?

variable toto cadrée à gauche, tronquée à 6 caractères :

```
typeset -L6 toto
```

variable toto cadrée à droite, tronquée à 6 caractères :

```
typeset -R6 toto
```

informations :

voir guide : les variables
la commande `typeset`

IV.1.9. Comment lancer mes propres commandes sans être obligé de spécifier leur chemin absolu?

Il faut mettre le répertoire où se trouve la commande dans la liste de la variable PATH

Si ma nouvelle commande se trouve dans le répertoire *repert :*

PATH=$PATH **:** *repert*

Si vous voulez avoir la variable PATH correctement positionnée à chaque session, placez l'affectation dans votre fichier *.profile*

IV.1.10.

informations :

voir guide : les variables
la localisation des commandes

IV.1.10. Comment rediriger les sorties standard et d'erreur vers des fichiers séparés?

C'est la mise en pratique des redirections.

exemple :

```
command 2>trace_std_erreur 1>trace_std_sortie
```

envoie la sortie d'erreur de la commande `command` dans le fichier `trace_std_erreur`, et la sortie standard dans le fichier `trace_std_sortie`.

informations :

voir guide : les fichiers, les descripteurs
les redirections

IV.1.11. Comment écrire un script qui ait 3 sorties standard?

Il faut que le programme écrive dans trois descripteurs différents.

IV.1.11.

1/ Exemple avec la commande `print`

```
$ more troisdes_1
#
# ce programme ecrit une ligne
# sur les descripteurs 3, 5, et 8
#
print "debut de traitement"
print -u3 "sortie sur le descripteur 3"
print -u5 "sortie sur le descripteur 5"
print -u8 "sortie sur le descripteur 8"
print "fin de traitement"
$ troisdes_1 3>sortie_3 5>sortie_5 8>sortie_8
debut de traitement
fin de traitement
$ more sortie_*
::::::::::::::
sortie_3
::::::::::::::
sortie sur le descripteur 3
::::::::::::::
sortie_5
::::::::::::::
sortie sur le descripteur 5
::::::::::::::
sortie_8
::::::::::::::
sortie sur le descripteur 8
$
```

IV.1.12.

2/ Exemple avec les redirections

```
$ more troisdes_2
#
# ce programme ecrit une ligne
# sur les descripteurs 4, 5, et 6
#
print "debut de traitement"
print "sortie sur le descripteur 4" >&4
ls toto* >&5 # sortie sur le descripteur 5
print "sortie sur le descripteur 6" >&6
print "fin de traitement"
$ troisdes_2 4>sortie_4 5>sortie_5 6>sortie_6
debut de traitement
fin de traitement
$ more sortie_*
::::::::::::::
sortie_4
::::::::::::::
sortie sur le descripteur 4
::::::::::::::
sortie_5
::::::::::::::
toto1.c
toto2.c
toto_base.c
::::::::::::::
sortie_6
::::::::::::::
sortie sur le descripteur 6
$
```

Vous pouvez remarquer que dans les deux cas, on peut en plus utiliser les sorties standards correspondant aux descripteurs 0, 1 et 2.

informations :

voir guide : les fichiers, les descripteurs
les redirections

IV.1.12. Comment dérouter l'exécution d'un programme sur réception d'un signal?

On suppose que le signal est *SIG*, et que la commande à exécuter (qui peut d'ailleurs être une fonction, ou plusieurs commandes séparées par des point-virgules, le tout encadré par des quotes) est `command`.

Il faut mettre en début de zone concernée :

`trap` *`command`* *SIG*

Il est possible de mettre plusieurs signaux pour un `trap`, ou de mettre plusieurs `trap` portant sur différents signaux.

IV.1.13.

La commande `command` doit être exécutable par le shell au moment de la réception du signal *SIG*.

informations :

voir guide : la commande `kill`
la commande `trap`
comment écrire un shell propre

IV.1.13. Comment effectuer une sélection parmi plusieurs solutions (CASE)?

On note <*exp_géné*> une expression générique.

Squelette :

case $*variable_a_tester* in
<*exp_géné1*>)
instruction1
instruction2
...
instructionN ; ;
<*exp_géné*2> | <*exp_géné*3> | <*exp_géné*4>)
instruction
... ; ;
esac

Les | entre expressions génériques signifient des OU logiques. On est bien sûr autorisé à mettre des constantes à la place des <*exp_géné*>.

Pour une *variable_a_tester* donnée, tous les tests successifs sont évalués jusqu'à ce qu'un d'entre eux soit valide; alors les instructions qui correspondent sont exécutées, puis l'interpréteur quitte le bloc `case-esac` : parmi toutes les tests, un seul peut être vérifié.

IV.1.13. Exemple :

```
$ more essai_case
#
# test du case avec le premier
# argument fourni au fichier
#
case "$1" in
    +([0-9]) )            print "c'est un entier";;
    toto | tata | tutu )  print "c'est un mot d'enfant";;
    A* )
                          if [[ $1 = AB* ]]
                              then print "ca commence par AB"
                              else print "ca commence par A"
                          fi ;;
    "" )                  print "il n'y a pas d'argument"
                          print "usage: essai_case <argument>";;
    * )                   print "valeur non repertoriee";;
esac
$
$ essai_case 123
c'est un entier
$ essai_case tutu
c'est un mot d'enfant
$ essai_case ABC
ca commence par AB
$ essai_case AC
ca commence par A
$ essai_case
il n'y a pas d'argument
usage: essai_case <argument>
$ essai_case ambigu
valeur non repertoriee
$
```

Notez que $*variable_a_tester*, c'est à dire $1 a été encadré avec des double-quotes ; c'est une précaution à prendre au cas où la variable $1 ne serait pas instanciée.

informations :

voir guide : les expressions conditionnelles
les expressions génériques

IV.1.14. Comment ôter le suffixe à un nom de fichier? IV.1.14.

Utiliser de préférence un opérateur sur une variable affectée avec le nom du fichier, sinon les commandes `basename` et `dirname`.

```
$ fichier=/tmp/prefixe.suf
$ echo ${fichier%.*}
/tmp/prefixe
$ basename $fichier
prefixe.suf
$ basename $fichier .suf
prefixe
$ echo ${fichier#*.}
suf
$ echo ${fichier##*/}
prefixe.suf
$ dirname $fichier
/tmp
```

informations :

voir guide : les opérateurs sur les variables

IV.1.15. Comment associer plusieurs conditions dans un test (IF)?

La seule règle est qu'il doit y avoir un “fi” pour chaque “if” ouvert. La condition “else” est optionnelle.

```
$ if [ -d ./exemples ]
> then print "./exemples est un repertoire"
>      if [ -w ./exemples ]
>      then print "autorise en ecriture"
>      else print "non autorise en ecriture"
>      fi
> else print "./exemples n'est pas repertoire"
> fi
./exemples est un repertoire
autorise en ecriture
$
```

informations :

voir guide : les expressions conditionnelles

les expressions génériques

IV.1.16.

IV.1.16. Comment se débarrasser de messages d'erreurs qui ne m'intéressent pas?

La quasi-totalité des commandes UNIX envoient leurs messages d'erreur sur la sortie d'erreur standard, c'est à dire le descripteur 2.

Il existe un fichier "poubelle", une sorte de trou sans fond, appelé `/dev/null` ; en fait, ce n'est pas un fichier physique, et les informations qu'on écrit dedans sont perdues.

Rajouter après la commande la redirection :

```
2>/dev/null
```

informations :

voir guide : les descripteurs de fichier
les redirections

IV.2. Cas types avec la ligne de commande

IV.2.1.

IV.2.1. Comment bénéficier des facilités de l'historique des commandes?

* vérifier que le shell que vous utilisez est bien le KornShell

`echo $SHELL` doit donner : `/bin/ksh`

* vérifier que les variables suivantes sont bien affectées :

```
HISTSIZE
HISTFILE
EDITOR
```

sinon les affecter (en les plaçant dans le fichier `.kshrc`) :

```
HISTSIZE=40
HISTFILE=~/.sh_history
EDITOR=vi
```

Chaque nouveau `ksh` sera alors correctement configuré pour l'historique.

* taper une commande, (suivie d'un <*return*>), puis sur la touche <*escape*>, puis sur la touche <*moins*>: vous devez retrouver sur la ligne en cours votre précédente commande.

informations :

voir guide : les variables prépositionnées

IV.2.2. Comment retrouver une commande contenant un mot particulier?

Pour retrouver la précédente commande, taper :

<escape> **-** (d'abord sur <escape>, ensuite sur -)

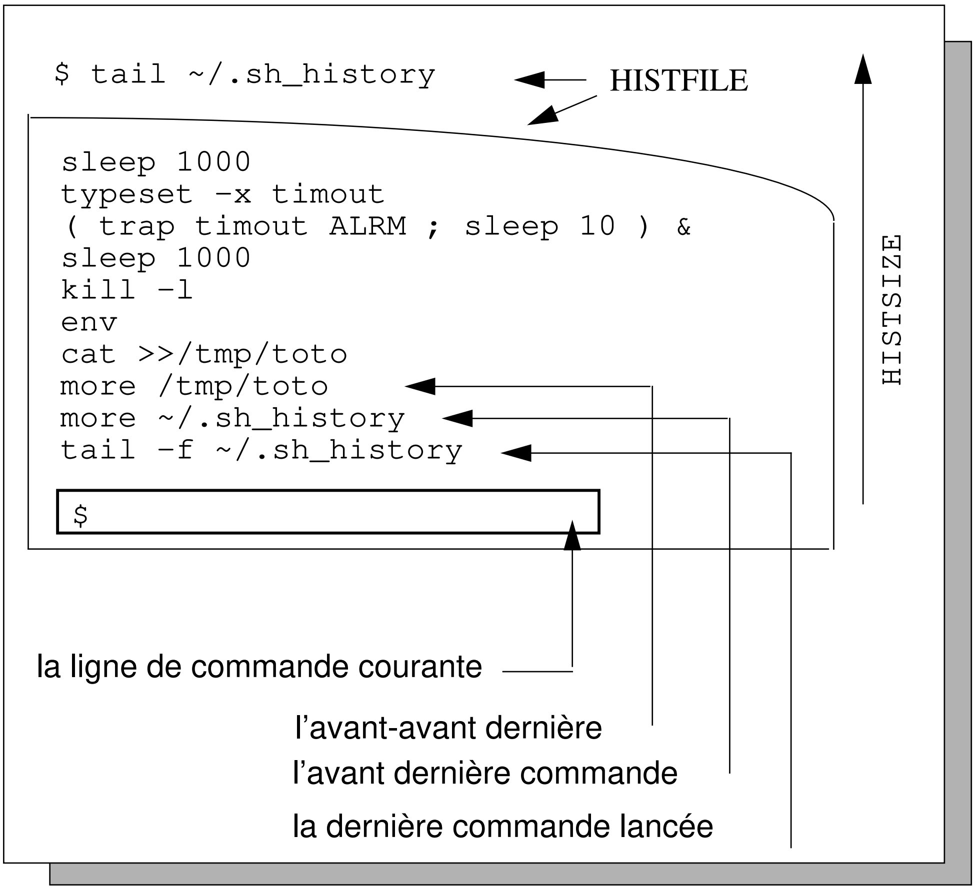
Pour retrouver la dernière commande lancée contenant "toto", taper :

<escape> /toto <return>

Explication :

Les différentes commandes que vous tapez sont placées par le shell dans le fichier indiqué par la variable HISTFILE. Le nombre maximum de commandes conservées est donné par le contenu de la variable HISTSIZE.

IV.2.2. La ligne de commande courante correspond à la dernière ligne du fichier, la précédente commande à l'avant dernière ligne du fichier, etc...



Lorsque votre curseur est devant le caractère `$` de la ligne de commande, il faut vous imaginer que vous venez de lancer `vi` sur le fichier d'historique, et que vous êtes en mode *insertion* sur la dernière ligne. Les lignes qui précèdent sont les commandes lancées précédemment. Donc, pour retrouver une commande, on procède comme sous `vi` :

- pour passer en mode *déplacement,* on tape <escape>

- pour remonter d'une ligne (i.e d'une commande), on tape ensuite - ou K : la commande sur laquelle on pointe dans le fichier HISTFILE prend place sur la ligne de commande courante.

- pour redescendre d'une ligne (i.e d'une commande), on tape + ou J

- pour repasser en mode insertion, on peut taper i ou a , etc...

- pour rechercher une chaîne, il faut être en mode *déplacement* (donc avoir tapé <escape>), puis, taper le caractère / suivi de la chaîne à rechercher, puis <return>. Une fois la première occurrence trouvée, on peut en obtenir une seconde en tapant `n`. La seule incohérence par rapport à `vi` est que la recherche se fait en remontant vers le début du fichier, mais sinon tout est pareil.

informations :

voir guide : les variables prépositionnées

référence `vi` : (voir la bibliographie)

IV.2.3. Comment étendre le nom d'un fichier que l'on est en train de taper? IV.2.3.

voir

voir

taper le début du nom, puis : <escape> <escape>

Détail :

Sur la ligne de commande, vous êtes en train de taper un nom de fichier. Arrêtez vous aux premiers caractères, puis tapez <escape><escape> ; le shell complète le nom, entièrement s'il n'y a qu'une possibilité dans le répertoire concerné. Sinon, il s'arrête à l'endroit du nom à partir duquel il existe plusieurs solutions.

IV.2.4. Pour avoir la liste des possibilités, tapez <escape>**=**, puis, tapez à nouveau <escape> et rajoutez le minimum de caractères pour lever l'ambiguïté, retaper <escape><escape> pour étendre le nom, etc...

voir

voir

```
$ ls
test1.bad test1.bet1 test1.bet2 test2.bad

$ ls *1*2
                         <escape><escape>

$ ls test1.bet2
                          <return>
test1.bet2
$
- - - - - autre exemple - - - - - -
$ echo te
                           <escape><escape>

$ echo test

1) test1.bad                <escape>=
2) test1.bet1
3) test1.bet2
4) test1.bet3
5) test1.res
6) test2.bad

$ echo test
                          <escape>

$ echo test

                            2<escape><escape>

$ echo test2.bad
```

informations :

référence `vi` : (voir la bibliographie)

IV.2.4. Comment lancer une commande sans avoir à attendre sa fin?

rajouter le caractère & après votre *commande*

Explication :

Si vous tapez une *commande* suivi de `<return>`, le shell ne réagit plus jusqu'à ce que la *commande* en cours soit terminée.

Pour reprendre la main de suite, il faut lancer la *commande* dans l'arrière-plan (ou "background") en rajoutant à la fin de la ligne le caractère &. IV.2.4.

Attention :

- si la *commande* que vous lancez en arrière-plan fournit des informations sur les sorties d'erreur ou standard, elles se feront vers les fichiers correspondant au moment du lancement : en particulier, la sortie d'erreur, si elle n'est pas redirigée, pointe sur la console, et risque fort de venir se mêler au texte que vous êtes en train de taper sur la ligne de commande, ou sous `vi`, à un moment inopportun. Donc, il est préférable de lancer la *commande* avec des redirections :

  `commande` `>resu 2>resu.err &`

- si la *commande* que vous lancez attend des informations en entrée, elle va se bloquer, sauf si vous avez spécifié une redirection en entrée.

Exemple :

voir

```
$ more cmd_bg
#
# test d'une commande en attente de deux
# valeurs sur l'entree standard
read val1
read val2
print $val1 $val2
$ cmd_bg &
[1] 29661
$
[1] + Stopped(tty output) cmd_bg &
$ jobs
[1] + Stopped(tty output) cmd_bg &
$ more cmd_bg.input
un
deux
$ cmd_bg <cmd_bg.input &
[2] 29663
$ un deux
$ echo "un\ndeux" | cmd_bg &
[3] 29664
[2] - Done cmd_bg <cmd_bg.input &
$ un deux
```

Les deux méthodes les plus simples à utiliser imposent en premier lieu d'anticiper le nombre d'entrées nécessaires pour répondre aux attentes de la commande. On place alors les réponses soit dans un fichier vers lesquels on redirige l'entrée standard

IV.2.5.

(première solution proposée ci-dessus), soit dans un `echo` pipé avec la commande (le passage à la ligne est symbolisé par le caractère `\n`)

informations :

voir guide : le caractère spécial `&`
la gestion des processus
les redirections

IV.2.5. Comment lancer une commande qui ne soit pas tuée si on se déconnecte?

```
nohup
at, batch
trap
```

nohup est une commande UNIX qui permet de lancer votre *commande* sans qu'elle soit tuée si vous vous déconnectez (elle inhibe la réaction au signal HUP envoyé lors de la déconnexion).

nohup *ma_commande* &

ma_commande doit être un nom de fichier binaire ou shell script exécutable. Les sorties standard ou d'erreur sont redirigées vers un fichier nommé `nohup.out`.Le `&` permet de reprendre la main de suite.

at et batch sont des commandes qui permettent aussi de lancer des commandes de manière détachée. Avec `batch`, la commande commence à s'exécuter tout de suite, alors que `at` permet de spécifier la date de début d'exécution.

```
echo ma_commande | batch
echo ma_commande | at 18 : 05
echo ma_commande | at now + 3 hours
```

permettent de lancer ma_commande respectivement de suite, à 18 heures 5 minutes, et dans 3 heures.

Pour lister les commandes précédemment programmées par `at` et en attente de d'exécution, taper :

```
at -l
```

Pour supprimer une commande programmée par `at` et en attente d'exécution, taper :

```
at -r numéro_de_job
```

numéro_de_job étant un terme de la forme xxxxxxx.a, que vous pouvez obtenir en listant les commandes par `at -l`.

trap

Si votre *commande* est un programme shell, vous pouvez aussi placer au début du programme la ligne :

`trap` "" HUP

et lancer la *commande* normalement ; votre programme continuera de tourner après la déconnexion.

informations :

voir guide : la gestion des processus
la commande `kill` et les signaux
la commande `trap`

IV.2.6. Comment se replacer dans le répertoire que l'on vient de quitter?

taper :

```
cd -
```

ou `cd $OLDPWD`

informations :

voir guide : les variables internes du shell

IV.2.7. Comment lister les variables positionnées? celles exportées?

Pour les variables positionnées dans le shell courant, taper

```
set
```

Pour les variables positionnées et exportées :

```
env
```

informations :

voir guide : les variables internes du shell
la commande `set`

IV.2.8. Comment mettre le répertoire courant dans le prompt?

PS1='$PWD $ '

informations :

voir guide : les variables internes du shell
le mode d' interprétation des commandes

IV.2.9. Comment écrire un alias?

`alias` *`mon_alias='ma_commande'`*

La commande suivante :

`mon_alias` `para1 para2 para3`

est transformée après substitution en :

IV.2.10.

`ma_commande para1 para2 para3`

Remarque : les paramètres passés à `mon_alias` sont recopiés dans le même ordre à la suite de la chaîne `'ma_commande'`, sans possibilité de mixer.

Pour que l'alias soit positionné dans chaque interpréteur de chaque fenêtre, placer la déclaration de l'alias dans votre fichier `.kshrc`

informations :

voir guide : le mode d'interprétation des commandes : les alias

où mettre les alias (voir ci-après)

IV.2.10. Dans quel fichier faut-il déclarer les variables et les alias pour les avoir positionnées à chaque session?

Il existe deux fichiers de configuration dans votre répertoire principal qui permettent de positionner variables et les alias :

`.profile` qui est exécuté une seule fois au moment de la connexion à une machine.

On y met les variables à exporter, le positionnement du PATH, la configuration de la console (caractère <backspace>, etc), en gros, tout ce qui doit être positionné au moment du login.

Il est conseillé également d'indiquer le nom du fichier de configuration du Ksh, en positionnant la variable ENV (voir ci-dessous).

`.kshrc` qui est exécuté au démarrage de chaque interpréteur de commandes.

On y positionne les variables non exportées, on y définit les alias et les fonctions.

Remarque: lorsque le ksh démarre, il évalue la variable ENV et charge la configuration du fichier indiqué par ENV: en standard, c'est toujours `.kshrc`, mais rien ne vous empêche de le changer.

IV.2.11. Comment lancer une commande avec le contenu du buffer de la souris?

La commande choisie doit accepter l'entrée standard comme source de données.

1/ choisir une fenêtre et lancer la commande avec les bonnes options, mais sans fichier de données ni redirection de l'entrée standard ; la commande doit se bloquer et rester en attente ;

2/ avec la souris, sélectionner une zone alphanumérique d'une fenêtre (noircir), puis vider le contenu du buffer dans la fenêtre où a été lancée la commande. Vous pouvez aussi taper des caractères au clavier, des retour-chariots, et répéter à plusieurs reprises cette étape ; IV.2.12.

3/ toujours dans la fenêtre où a été lancée la commande, taper un retour chariot (<return>), puis le caractère <control D> : la commande commence alors à s'exécuter.

informations :

voir guide : le contrôle des processus par le clavier

IV.2.12. Où trouver la liste des caractères ascii?

lister le fichier `/usr/pub/ascii`

IV.2.13. Comment lister les fichiers par ordre de taille?

```
ls -l| sort -n +4
```

la commande `sort` fait un tri sur le cinquième champ (les champs sont numérotés à partir de zéro) du fichier généré par la commande `ll` ;

le cinquième champ correspond à la taille du fichier.

l'option -n stipule que le classement se fait de manière numérique.

Pour avoir le tri par ordre décroissant :

```
ls -l | sort -nr +4
```

avec l'option `-r` comme 'reverse'

informations :

voir guide : la commande `sort`

IV.2.14. Que faire lorsque le shell refuse d'afficher le "prompt" et d'exécuter mes commandes après chaque retour chariot?

1er cas : vous n'avez pas refermé une quote ou double-quote de la précédente commande tapée, voire une boucle `for` ou une structure `if` : dans ce cas, chaque <return> doit provoquer le passage à la ligne avec affichage du caractère **>** en première position.

=> taper <control C>, puis récupérer la commande avec l'éditeur et vérifier la syntaxe.

2ème cas : la commande précédente attend ses données sur l'entrée standard par défaut, c'est à dire le clavier de votre fenêtre courante.

=> taper <control D> pour signifier à la commande la fin de fichier dans l'entrée standard : dans ce cas, les caractères déjà tapés à l'écran sont utilisés par la commande.

IV.2.15.

Méfiez vous cependant, car le shell lui-même considère <control D> comme la fin des commandes que vous êtes en train de taper: Si c'est lui qui récupère la séquence, il se termine et rend la main: vous perdez la fenêtre ou la connexion avec laquelle vous travailliez.

=> taper <control C> pour annuler la commande.

3ème cas : la commande précédente n'est pas terminée ;

=> attendre ou

=> taper <control C> pour stopper la commande en cours.

informations :

voir guide : le contrôle des processus par le clavier

IV.2.15. Pourquoi la sortie d'écran continue t-elle de défiler alors que je viens de taper plusieurs fois <*control*>C?

Cela arrive généralement avec des commandes qui génèrent des sorties assez importantes (les lignes défilent en grand nombre sur la fenêtre).

Généralement, l'affichage des données à l'écran prend plus de temps que la génération des informations par la commande : par suite, l'affichage par X11 utilise un buffer pour stocker l'information momentanément. Lorsque vous interrompez l'exécution d'une commande, X11 continue de vider le buffer et d'afficher les informations à l'écran.

Pour stopper le défilement, utilisez `<control S>`, puis `<control C>` pour interrompre la commande correspondante, puis `<control Q>` pour reprendre l'affichage normal.

informations :

voir guide : le contrôle des processus par le clavier

IV.3.1.

IV.3. Cas types avec les commandes UNIX

IV.3.1. Comment trier un fichier sur le 3ème champ?

sort

voir

```
sort +2 mon_fichier
```

ou, avec la nouvelle syntaxe :

voir

```
sort -k2, mon_fichier
```

Pour la commande `sort`, les champs d'un fichier sont repérés à partir du rang 0.

informations :

voir guide : la commande `sort`

IV.3.2. Comment trier un fichier sur plusieurs champs?

La commande `sort` trie par défaut sur le premier champ, puis, si plusieurs lignes ont le premier champ identique, trie ces lignes sur le deuxième champ, etc...

Rien n'empêche de modifier l'ordre de prise en compte des champs pour le tri : on peut commencer par trier sur le troisième champ, puis s'il y a ambiguïté, sur le quatrième, puis s'il y a encore ambiguïté, sur le premier ; cela donne (en notant les champs à partir de zéro) :

voir

```
sort -k2,4 -k0, mon_fichier
```

La séquence '`2,4`' indique que le tri est fait sur la séquence de champs numérotés de 2 à 4, 4 non inclus. Si le chiffre '4' n'était pas mis, le tri serait fait sur la séquence de champs numérotés de 2 jusqu'à la fin de ligne.

voir

Attention, ne pas oublier que les champs sont numérotés à partir de 0.

informations :

voir guide : la commande `sort`

IV.3.3. Comment rechercher un fichier dans mon arborescence?

find

Pour rechercher un fichier de nom donné dans *répertoire*, et dans les sous-répertoires :

```
find répertoire -name 'nom_du_fichier' -print
```

IV.3.4.

nom_du_fichier peut comprendre des caractères spéciaux du shell, comme * ou ?, etc...

Pour rechercher les fichier dont la taille est exactement *N* caractères :

```
find répertoire -size Nc -print
```

Pour rechercher tous les fichiers dont la taille est différente de *N* caractères :

```
find répertoire ! -size Nc -print
```

Pour rechercher les fichiers modifiés il y a moins de trois jours :

```
find répertoire -mtime -3 -print
```

informations :

voir guide : la commande `find`

IV.3.4. Comment exécuter une commande sur une liste de fichiers?

Les noms des fichiers à traiter sont présent sur l'entrée standard, ou dans un autre fichier (*liste*) :

xargs

```
cat liste | xargs ma_commande
```

`xargs` lance `ma_commande` pour chaque ligne du fichier d'entrée, en fournissant comme argument à *ma_commande* chacun des champs de la ligne.

Exemple :

```
ls -1 | xargs rm
```

exécute une commande `rm` pour chacun des fichiers listés par la commande `ls`.

L'option `-1` du `ls` force la sortie de la commande sur une colonne.

while do done

Dans un shell, on peut remplacer `xargs` (qui est pratique pour lancer une commande simple) par une boucle `while do done` si le traitement est plus complexe.

informations :

voir guide : la commande `xargs`
les structures de boucle
Comment lire un fichier ligne à ligne?

manuel (1) : la commande `xargs`

IV.3.5. Comment exécuter une commande sur tous les fichiers de mon arborescence?

IV.3.5.

La commande est à lancer sur tous les fichiers d'un répertoire et de ses sous-répertoires ;

find

`find` *`répertoire`* `-exec` *`ma_commande`* `{} \;`

exécute successivement *`ma_commande`* avec comme paramètre chacun des fichiers trouvés dans *`répertoire`* et ses sous-répertoires.

La séquence `{}` symbolise le nom du fichier courant (passé comme argument à *`ma_commande`*), et la séquence `\;` symbolise la fin de la commande

Exemple :

```
$ ls sort*
sortie_4 sortie_5 sortie_6
$ find . -name 'sort*' -exec ls -l {} \;
--w-r--r-- 1 logis users 28 Apr 30  1:04 ./sortie_4
-rw-r--r-- 1 logis users 28 Apr 30 14:04 ./sortie_5
-rw-r--r-- 1 logis users 28 Apr 30 14:04 ./sortie_6
$ find . -name 'sort*' -exec ls -l {} \; -exec echo {} "ok" \;
--w-r--r-- 1 logis users 28 Apr 30 14:04 ./sortie_4
./sortie_4 ok
-rw-r--r-- 1 logis users 28 Apr 30 14:04 ./sortie_5
./sortie_5 ok
-rw-r--r-- 1 logis users 28 Apr 30 14:04 ./sortie_6
./sortie_6 ok
$ find . -name 'sort*' -exec echo {} "ok" \;
./sortie_4 ok
./sortie_5 ok
./sortie_6 ok
$
```

informations :

voir guide : la commande `find`

manuel (1) : la commande `find`

IV.3.6. Dans un fichier, comment supprimer les lignes dont les 2 derniers champs sont identiques?

On recopie les lignes dont l'avant-dernier et le dernier champ ne sont pas égaux entre eux :

`awk 'NF>=2 && $(NF-1)!=$NF {print}'` *`mon_fichier`*

IV.3.7. Pour la commande `awk`, `$i` correspond à la valeur du champ numéroté `i` (en commençant au rang 1), et la variable NF indique le nombre de champs dans la ligne : donc, $NF correspond au dernier champ de la ligne, et $(NF-1) à l'avant dernier.

La commande revient à imprimer (`{print}`) les lignes qui ont au moins deux champs (`NF>=2`) et (&&) dont les deux derniers champs sont égaux (`$NF==$(NF-1)`).

La présence de la commande print (`{print}`) est optionnelle, car c'est l'action réalisée par défaut.

informations :

voir guide : la commande `awk`

manuel (1) : la commande `awk`

IV.3.7. Dans un fichier trié, comment conserver une seule ligne parmi celles dont le 2ème champ est identique?

Cela consiste à trier le fichier sur le deuxième champ, puis à comparer les lignes consécutives pour supprimer celles dont le deuxième champ est identique à celui de la ligne qui précède.

Exemple :

```
$ more tst_uniq
un deux trois quatre cinq
un 2 trois quatre cinq
un deux trois 4 cinq
un deux trois quatre cinq
un deux trois quatre 5
$ sort +1 tst_uniq |
> awk 'chp2 != $2 { print      }
>                 { chp2 = $2 }'
un 2 trois quatre cinq
un deux trois 4 cinq
$
```

Explications :

```
sort +1 tst_uniq
```

on trie le fichier `tst_uniq` sur le deuxième champ

```
awk 'chp2 != $2 { print      }
                { chp2 = $2 }'
```

* pour chaque ligne où le contenu de la variable `chp2` n'est pas égal au deuxième champ, on réalise l'action {print}, c'est à dire l'affichage de la ligne.

* quelque soit la ligne, on réalise l'affectation `chp2 =` <deuxième champ>. Cela revient à mémoriser le deuxième champ pour le test qui aura lieu à la ligne suivante.

Remarques :

- pour `awk`, une variable non affectée vaut la chaîne nulle : c'est ce que vaut `chp2` à la première évaluation.
- les champs sont numérotés à partir de 1 pour la commande `awk`.

informations :

voir guide : la commande `sort`
la commande `awk`

IV.3.8. Comment convertir un nombre décimal en hexa?

1ère solution : typage d'une variable du shell

```
$ typeset -i16 val
$ val=35
$ echo $val
16#23
$ echo ${val#*#}
23
```

2ème solution : la commande `dc`

```
$ val=35
$ echo "16o${val}p" | dc
23
$
```

on passe à la commande `dc` :

* le terme `16o` qui indique que la sortie se fait en hexa
* le contenu de la variable `val` (`${val}`)
* le terme p qui provoque l'affichage dans la base spécifiée.

informations :

voir guide : la commande interne `typeset`
la commande `dc`, la commande `bs`
les opérateurs sur les variables

IV.3.9.

IV.3.9. Comment lancer une commande à une date donnée?

at

Pour lancer *ma_commande* à 21 heures 35 aujourd'hui :

```
echo ma_commande | at 21 : 35
```

Pour lancer *ma_commande* d'ici 3 heures :

```
echo ma_commande | at now + 3 hours
```

Pour lancer *ma_commande* d'ici 3 jours, à la même heure :

```
echo ma_commande | at now + 3 days
```

informations :

manuel (1) : la commande `at`

IV.3.10. Comment retrouver les fichiers qui n'ont pas été modifiés depuis plus de trois jours? depuis moins de 3 jours?

plus de 3 jours :

```
find . -mtime +3 -print
```

pour chaque fichier, le test `-mtime +3` est réalisé, et s'il est valide, la commande `-print` est lancée.

moins de 3 jours :

```
$ find . -mtime -3 -exec ls -ld {} \;
drwxr-xr-x 2 logis users 1024 May 14 17:07 .
-rw-r--r-- 1 logis users   30 May 14 14:40 ./tst_sort1
-rw-r--r-- 1 logis users   51 May 14 14:22 ./tst_sort
-rw-r--r-- 1 logis users  119 May 14 18:03 ./tst_uniq
-rw-r--r-- 1 logis users   99 May 14 17:14 ./tst_3chp
$
```

pour chaque fichier, le test `-mtime -3` est réalisé, et s'il est positif, la commande `-exec ls -ld {} \;` est lancée, ce qui provoque l'exécution du `ls -ld` avec le nom du fichier courant (symbolisé par { }).

informations :

voir guide : la commande `find`

IV.3.11.

IV.3.11. Comment extraire les 4ème et 5ème champs des lignes de mon fichier?

awk

```
awk '{ print $4, $5}' mon_fichier
```

cut

```
cut -d" " -f4,5 mon_fichier
```

l'option -d"<space>" spécifie que le séparateur de champs est le caractère <space> (<tab> par défaut)

informations :

voir guide : la commande `awk`

manuel (1) : la commande `awk`
la commande `cut`

IV.3.12. Comment garder toutes les lignes de mon fichier qui ne contiennent pas une chaîne particulière?

grep

```
grep -v 'ma_chaine' mon_fichier
```

l'option -v spécifie que les lignes affichées sont celles qui ne contiennent pas *ma_chaine*.

awk

```
awk '! /ma_chaine/' mon_fichier
```

le caractère ! correspond à un NON logique. Si la chaîne *ma_chaine* n'apparait pas dans la ligne courante, elle est affichée (l'action par défaut est {`print`})

informations :

voir guide : la commande `grep`
la commande `awk`

IV.3.13. Comment recopier un répertoire entier?

cp

La commande cp peut être utilisée en mode "récursif" avec l'option `-r`.

```
cp -r ancien nouveau
```

Si nouveau n'existait pas, il est créé, et son contenu est celui d'ancien.

IV.3.13. Si nouveau existait déjà, une copie d'ancien est placée dans nouveau, sous le même nom.

Avantage :

Simplicité.

Inconvénients :

- les dates de tous les fichiers du nouveau répertoire sont les dates actuelles, et non pas celles des fichiers d'origine.

- les liens symboliques ne sont pas recopiés, et les liens physiques sont recopiés mais pas en temps que lien (le résultat est un fichier contenant la même chose que le lien, mais ce n'est plus un lien).

tar

```
cd ancien
tar cf - . | ( cd nouveau ; tar xvf - )
```

Le répertoire nouveau doit avoir été créé avant de lancer la commande.

Avantages :

- les dates de fichiers sont celles des fichiers originaux.

- comme il y a un pipe, on peut faire passer facilement le tout par le réseau ; exemple :

voir

```
cd ancien
tar cf - . | remsh machine "cd nouveau ;tar xvf -"
```

Idem, précédent, mais la recopie s'effectue sur `machine`, plutôt que sur la machine sur laquelle vous êtes connecté.

- les liens symboliques et physiques sont conservés.

Inconvénients :

- complexité

- les dates de fichiers sont conservées, mais pas les dates de répertoires, qui sont celles de création du nouveau répertoire.

cpio

```
cd ancien
find . -depth -print | cpio -pd nouveau
```

Le répertoire nouveau doit avoir été créé avant de lancer la commande.

Avantages :

- les dates de fichiers et répertoires sont celles des fichiers originaux.

- les liens symboliques et physiques sont conservés.

IV.3.14.

Inconvénients :

complexité

informations :

voir guide : les liens

manuel (1) : les commandes `cp, cpio, tar`

IV.3.14. Comment retrouver les fichiers qui font un lien sur un fichier donné?

find

```
$ ln -s tst_uniq tst1
$ ln tst_uniq tst2
$ ls -l tst1 tst2
lrwxr-xr-x 1 logis users   8 May 15 11:56 tst1 -> tst_uniq
-rw-r--r-- 2 logis users 119 May 14 18:03 tst2
$ find . -linkedto tst_uniq -print
./tst_uniq
./tst2
$ find . -follow -linkedto tst_uniq -print
./tst_uniq
./tst1
./tst2
$
```

voir

voir

Explication :

La première commande `ln` crée un fichier `tst1` qui est un lien symbolique vers `tst_uniq` ; la deuxième crée un fichier `tst2` qui est un lien physique vers `tst_uniq`.

La première commande `find` recherche uniquement à partir du répertoire courant (symbolisé par `.`) les fichiers liés physiquement à `tst_uniq` (même inode).

Le deuxième `find` comporte l'option `-follow` qui fait suivre les liens symboliques : les noms des fichiers obtenus sont ceux liés physiquement ou symboliquement au fichier `tst_uniq`.

IV.3.15. Une autre méthode consiste à travailler avec le numéro d'inode qui référence de manière unique un fichier.

```
$ ls -i tst_uniq
 51351 tst_uniq
$ find . -inum 51351 -print
./tst_uniq
./tst2
$ find . -follow -inum 51351 -print
./tst_uniq
./tst1
./tst2
$
```

On commence par obtenir le numéro d'inode du fichier de référence, puis on parcourt l'arborescence à la recherche de tous les fichiers ayant le même inode. La première commande donne les fichiers étant liés physiquement, la seconde les fichiers étant liés physiquement ou logiquement.

L'option `-linkedto` du `find` fait en fait automatiquement le lien entre le nom du fichier et l'inode: le principe de la recherche est en fait le même.

informations :

voir guide : la commande `find`
les liens, la commande `ln`

IV.3.15. Pourquoi je n'arrive pas à supprimer mon fichier? à en créer un?

Pour créer un fichier ou le supprimer, il faut avoir les droits d'écriture sur le répertoire qui contient le fichier.

Donc, tapez la commande `ls -ld` *`le_répertoire`*, puis la commande `id` et comparez les droits autorisés, le propriétaire, le groupe du répertoire avec vos noms d'utilisateur et de groupe :

- le nom de propriétaire et votre nom d'utilisateur doivent correspondre si le répertoire a les droits en w pour le propriétaire,
- ou alors les deux noms de groupe si le répertoire a les droits en w sur le groupe,
- ou alors le répertoire doit avoir le droit w pour les autres ("others")

Remarque :

Si vous essayez de créer un fichier qui, en fait, existe déjà, le propriétaire, le groupe, et les droits de l'ancien fichier sont conservés. Par exemple, si vous faites *ma_commande* >/tmp/toto et que le fichier /tmp/toto existe déjà et n'a pas les bons droits pour vous, vous n'arriverez pas à y écrire dedans.

informations :

voir guide : les droits sur les fichiers

IV.3.16. Comment positionner par défaut les droits d'un fichier ?

Lorsque on crée un fichier, ses droits[1] sont positionnés en fonction de valeurs par défaut.

On peut consulter cette valeur par défaut en tapant la commande `umask`, ou la modifier si on fournit une valeur en paramètre.

Le nombre qui est donné en résultat, ou que l'on donne en paramètre, est composé de trois chiffres, le premier s'appliquant aux droits de l'utilisateur, le second à ceux du groupe, le troisième à tous les autres utilisateurs.

Ce nombre indique, pour chaque type d'utilisateurs, les droits à *enlever* en lecture, écriture, et exécution.

Numériquement, le droit en lecture vaut 1, celui en écriture vaut 2, et celui en exécution vaut 4.

Si le résultat de la commande `umask` donne `022`, cela indique qu'à toute nouvelle création de fichier, les droits d'écriture (valeur = 2) pour le groupe et les tous les autres sont supprimés.

Le fichier est donc du type `-rw-r--r--`

Si vous désirez que vos fichiers ne soient autorisés en lecture que pour vous seul, il faut modifier la valeur par défaut en tapant la commande `umask 066` : on enlève alors les droits de lecture (valeur = 2) et les droits en écriture (valeur = 4), au total 6, pour le groupe (deuxième position) et tous les autres (troisième position).

Toutes les combinaison sont possibles.

Les droits par défaut peuvent être positionnés différemment dans les différents shells en activité: ce qui veut dire aussi que si vous modifiez les droits dans un shell, ils seront inchangés dans les autres. C'est pourquoi la meilleure méthode est de placer la commande `umask` dans le fichier `.profile`: la valeur est vue ensuite partout, par héritage.

La commande `chmod` permet ensuite, au coup par coup, de changer les droits de fichiers déjà créés, suivant le même principe.

1. Voir les droits sur les fichiers, page 23

IV.3.17. ***informations*** :

voir guide : les droits sur les fichiers

manuel(1): les commandes `chmod` et `umask`

IV.3.17. Comment déterminer le type ou le contenu d'un fichier?

file

strings

what

Il est facile de déterminer le contenu d'un fichier texte (programme shell, source ADA ou C, données, etc...) en le listant par une commande classique (`more`, `vi`, etc), mais plus délicat de découvrir ce que renferme un fichier binaire.

En premier lieu, on peut appliquer la commande `file` sur le fichier de type inconnu :

- l'indication donnée mentionne un fichier de type `text` : se méfier, car l'analyse est souvent erronée (au moins 50% d'incertitude). Mais on peut en théorie lister son contenu.
- l'indication donnée mentionne un fichier de type `executable` : votre fichier est un binaire exécutable (100% exact).

Une autre exploration peut être faite en utilisant la commande `what` qui recherche dans le fichier une chaîne de caractères précédée de la séquence :

voir

`@(#)`

C'est le cas de pratiquement tous les exécutables UNIX qui intègrent ainsi leur version, ou leur date de mise à jour[1]. Mais, malheureusement, la plupart des applications ou des commandes qui ne font pas partie intégrante du système lui-même ne comportent pas une telle indication.

Si le fichier est un exécutable, ou un `core`[2], il est possible tout de même d'estimer son contenu en appliquant dessus la commande `strings`, qui liste toutes les constantes de type chaîne inclues dans le binaire : c'est en général suffisant pour déterminer le type de langage dont il est issu (voir les exemples avec la description de la commande `strings`).

informations :

voir guide : la commande `file`

la commande `strings`

1. un exemple est donné sur la commande UNIX `awk`; voir page 94.
2. voir le chapitre sur les messages d'erreur: page 208

IV.3.18. Comment avoir la réponse à une question qui n'apparait pas ci-dessus? IV.3.18.

L'information la plus fiable se trouve dans le manuel en ligne, par l'intermédiaire de la commande `man`. (chaque constructeur propose au travers de son environnement graphique une aide intéractive un peu plus élaborée, bien que l'information elle-même soit identique).

Faire `man` *`la_commande`* pour avoir des informations sur *`la_commande`*, et `man ksh`, pour avoir des informations sur le shell (variables, syntaxe des boucles, etc...).

En plus des information sur les commandes, vous pouvez vous documenter sur le format de certains fichiers.

voir

voir

Voici quelques entrées intéressantes:

- `man profile`: donne le format du fichier et quelques exemples.
- `man ascii` : liste des caractères ascii (idem /usr/pub/ascii)
- `man regexp` : définition des expressions régulières et des expressions génériques.
- `man suffix` : convention des suffixes sous Unix

Si vous possédez un manuel "papier", cherchez dans le manuel de référence section 1 ; les informations sont celles du manuel en ligne, sauf si la version de la documentation ne correspond pas à celle du système d'exploitation de votre machine.

Le manuel est écrit, c'est vrai, de manière assez rébarbative, mais vous y trouverez quasiment toutes les informations dont vous pouvez avoir besoin.

Vous pouvez aussi vous rapporter à des ouvrages de librairie, utiles pour avoir une idée des diverses possibilités du shell, mais qui ne tiendront que rarement compte des spécificités de la version d'UNIX que vous utilisez. (voir la bibliographie)

V.

V. Optimiser un programme shell

V.1. Coût d'exécution

IV.3.18.

Lorsqu' un programme s'exécute, les deux actions les plus coûteuses en ressources sont :

la création d'un processus

Lancer une commande UNIX (cp, ls, sort, vi...), ou un programme shell script, nécessite la création d'un processus.

Or, la création d'un processus demande un grand nombre d'opérations : faire de la place en mémoire, charger l'exécutable, mettre à jour toutes les tables internes (fichiers standard, signaux, etc..), vérifier les droits, ... Et lorsque le processus se termine, une partie de ces opérations est à reprendre...

Si la commande travaille de plus sur une petite quantité de données, le temps passé par la machine à créer, puis supprimer le processus est très important par rapport au temps passé sur la tâche réellement liée à la commande de l'utilisateur.

Exemple : .

```
$ time ( echo Durand | sed 's/D/d/' )
durand

real 0m0.08s
user 0m0.02s
sys 0m0.06s
$
```

la commande `time` mesure le temps d'exécution d'une autre commande :

`real` est le temps qui s'est écoulé entre le début et la fin de la commande : `echo Durand | sed 's/D/d/'`

`user` est le temps passé par le cpu à travailler pour l'utilisateur

`sys` est le temps passé par le cpu pour gérer le processus lui-même (création, etc..) et manipuler les ressources liées aux fichiers ouverts (pointeurs, index, etc...)

(la somme de `user` et de `sys` n'est jamais égale à `real` puisque le temps cpu est partagé entre les différents utilisateurs)

Vous pouvez remarquer que la commande :

`echo Durand | sed 's/D/d/'`

a consommé ici trois fois plus de ressources système que de ressources utilisateur.

Cet exemple montre qu'il existe bien un coût fixe lors du lancement d'une commande.

Si cette commande travaille sur un gros fichier, ce coût fixe est négligeable.

IV.3.18. Par contre, l'exemple indiqué ci-dessus, qui apparait souvent dans des scripts, parce que c'est vrai qu'il repose sur des commandes bien connues, est assez contestable :

- Si votre script comporte quelques exécutions de ce type en tout et pour tout, la simplicité et la lisibilité l'emportent sur un gain de performances finalement minime au total ;
- Mais très souvent, cette commande composée apparait dans des boucles `while`, par exemple sur des lectures de fichier. Il suffit que le fichier fasse 500 lignes et elle sera donc lancée 500 fois. Comme ici, la commande composée lance 2 processus pour se réaliser (un `echo` et un `sed`), c'est donc 1000 processus qui seront crées durant l'exécution de ce script.

Si c'est la seule méthode existante pour aboutir à la solution désirée, vous n'avez pas le choix.

Mais le plus souvent, il est possible de remplacer les *n* lancements de commande sur un petit peu d'information, par un seul lancement sur *n* fois un petit peu d'information.

Le principe consiste à :

- regrouper la totalité des données de départ dans un fichier ou dans un pipe
- sélectionner la partie, c'est à dire la ou les colonnes du fichier (ou du pipe) dans laquelle les données sont à traiter, et stocker le reste dans un fichier (ou pipe) annexe
- lancer la commande sur l'ensemble des données sélectionnées
- rassembler le tout

se qui se traduit en pratique par à peu près autant de commandes que d'opérations décrites ci-dessus : c'est à dire quatre ou cinq, et quelque soit la taille du fichier d'entrée.

Un exemple est donné ci-après.

<u>la création d'un fichier</u>

Comme on l'a indiqué ci-dessus, les fichiers sont manipulés et contrôlés via le système d'exploitation : chaque fois que vous créez ou supprimez un fichier, de nombreux index et tables sont mis à jour ; or la plupart du temps, ces informations se trouvent sur le disque, dont le temps de réponse est très important.

Créer ou supprimer un fichier, de la même façon que pour un processus, n'est rentable que si son contenu est conséquent.

C'est donc du vrai gaspillage que de sélectionner une ligne ou une chaîne de caractère et de la stocker dans un fichier : mettez la plutôt dans une variable !

V.2. Quelques conseils en vrac

IV.3.18.

• Utilisez de préférence les commandes internes du shell (en particulier, les opérateurs sur les variables) à des commandes équivalentes UNIX : comme il n'y pas de nouveau processus à créer, l'exécution est beaucoup plus rapide.

• les fonctions écrites en shell sont moins rapides que les commandes internes, mais plus rapides que les commandes UNIX.

• utilisez l'opérateur **[ ]** ou **[[ ]]** dans les tests `if` ou `while` plutôt que la commande UNIX `test`.

• il est plus rapide de faire exécuter une commande complexe (avec de nombreuses options) que plusieurs simples enchaînées (en raison du coût fixe de création des processus).[1]

• pour accélérer le démarrage de commandes UNIX très souvent appelées, il vaut mieux indiquer leur chemin d'accès absolu, plutôt que de laisser le shell les localiser dans le PATH.

• placez les répertoires contenant les commandes les plus souvent utilisées au début du PATH.

• Evitez d'avoir un PATH trop long.

• limitez dans le PATH la présence de répertoires vus par NFS (Network File System): à chaque démarrage (chaque console, chaque fenêtre), le shell tente d'accéder à chacun des répertoires du PATH, et si un des serveurs, ou le réseau, est indisponible, vous serez bloqué le temps du timeout (de l'ordre de la minute). Placez plutôt l'extension du PATH dans un script qui démarre l'utilitaire du serveur distant.

• spécifiez l'option `trackall`[2] qui génère un alias interne à chaque nouvelle commande lancée et cherchée dans le PATH : au lancement suivant, la localisation n'est plus à refaire.

• limitez les lectures de fichier ligne à ligne en utilisant des boucles `while-do-read-done` : le nombre de commandes lancées à l'intérieur de la boucle doit être diminué au maximum.

• si vous utilisez à chacun des cycles du traitement un fichier temporaire, inutile de le supprimer chaque fois : il suffit d'y écrire en mode *write* (avec une redirection **>**), ce qui le réinitialise, sans avoir à le recréer.

• évitez l'appel à des commandes UNIX à l'intérieur d'un programme `awk`[3].

1. cette remarque est à moduler en fonction de la quantité des données à traiter : sur de gros fichiers, il est parfois plus intéressant de lancer plusieurs commandes élémentaires mais d'exécution rapide, plutôt qu'une seule très complexe et donc plus lente.
2. `set -o trackall` placé en début de programme
3. par l'intermédiaire de la fonction `system`

V.3.1.

V.3. Exemples pour éviter les boucles

V.3.1. Recherches de chaînes

Un exemple classique consiste à manipuler deux fichiers de la façon suivante :

- un premier fichier `reference` contient des données de référence. Par exemple :

Jef	17623
Jean-Hugues	16173
Agnès	16579
Denis	16239
...	

qui contient une correspondance entre des noms et des numéros de téléphone.

- un deuxième fichier `travail` contient des données de travail. Par exemple :

Agnès	"inviter au restaurant"
Denis	"proposer un squash"
Jacques	"annuler le rendez-vous"
...	

qui contient une liste de personnes et un message à leur transmettre.

L'objectif est de créer un fichier résultat, qui contienne les noms des personnes à appeler, le message correspondant et le numéro de poste.

Première méthode avec une boucle :

```
exec 3<travail
while read -u3 nom message
do
        poste=$(grep "^$nom " <reference)
        print $nom $poste $message
done
```

Si on analyse l'exécution, on s'aperçoit :

- que la commande `grep` est lancée autant de fois qu'il y a de lignes dans le fichier `travail`
- que le fichier `reference` est parcouru lui aussi par le `grep` autant de fois qu'il y a de lignes dans le fichier `travail`.

Si les fichiers `travail` et `reference` sont très gros, on voit bien que le traitement est loin d'être optimisé.

Deuxième méthode sans boucle

```
sort -o reference.trie reference
sort -o travail.trie travail
join -1 1 -2 1 travail.trie reference.trie
```

Ici, on utilise pour faire la correspondance la commande `join`[1] et deux commandes `sort` pour trier les fichiers de départ sur le champ qui sert de jointure entre les deux.

En tout et pour tout, trois commandes, quelque soit la taille des fichiers d'entrée. Les commandes `sort` et `join` sont d'une complexité plus grande que `grep`, bien sûr, mais le gain est cependant énorme.

Bien sûr, le cas présenté ici est trivial, mais la démarche générale est toujours la même : elle consiste à regrouper deux fichiers d'entrée qui contiennent une partie des informations chacun, en un nouveau qui ne contient plus que les informations utiles, si possible rangées de manière homogène sur une même ligne ou une même colonne.

On réitère l'opération plusieurs fois jusqu'à avoir un fichier qui contienne toutes les informations utiles pour générer le résultat[2].

Notez que beaucoup de boucles `while-do-read-done` peuvent ainsi être remplacés *efficacement* par un enchaînement d'un petit nombre de commandes : `awk`, `sed`, `sort`, `join`, `paste`.

V.3.2. Manipulation d'une partie de ligne

L'exemple que l'on prend ici est celui d'un fichier (`travail`) qui sur chaque ligne contient deux valeurs en hexadécimal :

Par exemple :

Jef	125	f235
Totof	d12	10a4
Agnes	394b	12
Jean-Hugues	b456	312
Denis	0	8e97
...		

Le but de l'opération consiste à rajouter en fin de ligne la racine carrée de la somme des deux valeurs déjà existantes.

1. voir son utilisation en détail dans le chapitre sur les commandes UNIX: page 116
2. attention quand même à la taille du fichier : il ne s'agit pas d'en fabriquer un énorme !

V.3.2.

Première méthode avec une boucle :

voir

```
exec 3<travail
while read -u3 nom val1 val2
do
    val3=$(echo "ibase 16\nsqrt(0$val1+0$val2)"|bs)
    print $nom $val1 $val2 $val3
done
```

L'exécution de cette boucle provoque la création de deux processus par ligne de calcul : un `echo` et un `bs.`

(Le zéro rajouté devant les nombres avant le calcul évite à `bs` de confondre les nombres hexa commençant par une lettre avec une variable)

Deuxième méthode sans boucle

```
awk 'BEGIN{ print "ibase 16" }
     {
            print "sqrt(0" $2 "+0" $3 ")"
     }' <travail |
bs >tempo
paste travail tempo
```

Ici, on crée un fichier temporaire (en fait, un pipe) contenant toutes les données dans le format adéquat pour un calcul simultané par `bs` de toutes les valeurs. Puis, après passage par `bs`, on concatène ligne à ligne avec un `paste` le fichier résultat avec celui de départ, les deux fichiers ayant logiquement chacun le même nombre de lignes.

Quelque soit le nombre de lignes du fichier d'entrée, le calcul nécessite trois commandes (`awk`, `bs`, et `paste`) et un fichier temporaire.

V.3.2.

VI.

VI. Ecrire un programme shell propre

VI.1. Pourquoi perdre du temps?

V.3.2.

Chaque fois que vous passez du temps à écrire un programme shell pour une utilisation donnée, pensez qu'il y a de fortes chances pour qu'il soit utilisé plusieurs fois, dans des conditions vraisemblablement légèrement différentes, et alors que vous aurez oublié exactement la façon de l'utiliser. Il se peut aussi que d'autres personnes soient amenées à l'utiliser.

Dans tous les cas, vous éviterez beaucoup de fausses manipulations et de recherches inutiles par la suite si vous prenez quelques précautions au moment de sa création.

V.3.2.

VI.2. Choix du shell à utiliser

La version d'UNIX que vous utilisez comporte trois shells spécifiques : le sh, le csh, et le ksh que vous utilisez probablement. Comme leur syntaxe présente des particularités, un programme écrit pour un shell a très peu de chances de fonctionner correctement avec un autre.

Pour imposer un shell particulier lors de l'exécution d'un programme, il faut rajouter en première ligne :

```
#! /bin/mon_shell
```

mon_shell pouvant prendre les valeurs : `sh, csh` ou `ksh` suivant la syntaxe choisie. Le commentaire doit démarrer au début de la ligne.

Il vous est conseillé d'utiliser le `ksh`, celui-ci offrant tous les services des deux autres, plus quelques-uns supplémentaires...

VI.3. Où mettre le nouveau programme

V.3.2.

Sauf dans certains cas particuliers, il vaut mieux placer la version au point de votre programme dans un répertoire où seront réunis tous vos scripts.

Par exemple, vous pouvez créer un répertoire `bin` dans votre répertoire principal, et y mettre tous vos programmes à jour.

Pour lancer vos programmes depuis n'importe quel répertoire, rajoutez dans votre fichier de configuration `.profile` la modification adéquate de la variable PATH :

```
PATH=$PATH:$HOME/bin
```

V.3.2.

VI.4. Commentaires, version

Il y a de fortes chances pour qu'après quelques temps, vous ne sachiez plus exactement ce que fait votre programme, ni dans quelles conditions on peut l'utiliser.

N'hésitez pas à mettre quelques lignes de commentaires (commençant par un caractère `#`) qui aident beaucoup.

L'expérience montre que rapidement co-existent plusieurs versions du même programme, à des stades de développement différents, ou destinés à des usages légèrement différents : si vous n'avez pas mis de commentaires, il devient très difficile de retrouver l'usage exact de `prog,` par rapport à `prog1`, ou `prog%`, ou encore `prog_ok`, si leur contenu se ressemble un peu.

Il arrive aussi qu'un même programme soit recopié sur plusieurs machines, ou dans plusieurs répertoires ; si la date de dernière modification n'apparait pas dans les commentaires, vous aurez certainement des doutes sur la version de référence.

En résumé :

Il est bon de rajouter en début de programme (après le #! /bin/ksh) :

- le nom du programme
- à quoi il sert
- la date de dernière modification
- un exemple de commande complète (avec paramètres, options, etc...)
- éventuellement, certaines particularités....

<u>Remarque:</u>

Ce n'est pas l'objet de ce document, mais sachez qu'il est très utile, lorsque vous écrivez des programmes en langage C, Ada, etc.. d'insérer une chaîne de type constante, démarrant par

@(#)

et suivie des informations indiquées ci-dessus. En effet, une fois votre source compilé, vous pourrez retrouver ces informations en lançant la commande `what` sur votre exécutable ou votre module objet.

Sans quoi, vous n'aurez que la date de dernière modification, la taille, ou le nom du fichier pour retrouver la version considérée, ce qui est bien moins pratique et <u>sûr</u> qu'un commentaire en clair !

V.3.2.

VI.5. Test des options, vérification des arguments

Lorsque vous créez un programme, vous pouvez lui donner une certaine modularité en le paramétrant grâce à l'usage des paramètres ($1, $2, $3, etc...) éventuellement sous la forme d' options (-*opt1*, -*opt2*, etc...) spécifiées au lancement.

Si la validité de ces paramètres ou ces options n'est pas correctement testée, vous risquez fort après quelques temps d'oublier les conditions exactes de fonctionnement, de lancer de manière incorrecte votre programme et d'obtenir un résultat erroné, voire catastrophique.

De plus, il est particulièrement désagréable en retour d'une commande d'avoir un message d'erreur du shell souvent incompréhensible à premier abord (ce sera vraisemblablement le cas si vous passez à votre programme un type d'argument non prévu), plutôt qu'une indication pertinente du mauvais usage que vous en avez fait.

Il faut donc tester la validité de chacun des arguments passés au programme.

Le test se déroule en deux passes :

- On commence par vérifier que les options sont bien celles attendues et que les paramètres sont en nombre suffisant ; c'est une sorte d'analyse syntaxique.
- Ensuite, on vérifie que les valeurs recueillies sont pertinentes : nombres dans l'intervalle attendu, test d'existence du fichier si un paramètre correspond à son nom, etc...

VI.5.1.

VI.5.1. Test de la présence des options et paramètres

Exemple : on écrit un programme que l'on peut éventuellement lancer avec l'option `-opt1`, éventuellement avec l'option -opt2 suivie d'une valeur, et avec un paramètre obligatoire. Voici une façon de tester la présence des options et du paramètre : VI.5.1.

```
#! /bin/ksh

#
# exemple de test des options et arguments
# 15/05/92

option1=0
option2=0
val2=""
fic=""

usage="$0:  usage:  prog [-opt1] [-opt2 val] fichier"
while [ $# != 0 ]
do case $1 in
 -opt1 ) # option No 1
         option1=1 ; shift
         ;;
 -opt2 ) # option2 (suivie par la valeur associe)
         option2=1 ; shift
         if [[ "$1" = -* ]] || [ -z "$1" ]
         then print -u2 "$usage"
              exit 1
         else val2=$1 ; shift
         fi
         ;;
 -* )   # option inconnue
        print -u2 "$0: ($1) option inconnue"
        print -u2 "$usage"
        exit 1
        ;;
 * )    # le parametre en fin de commande
        if [ $# != 1 ]
        then print -u2 "$usage"
             exit 1
        else fic=$1; shift
        fi
        ;;
   esac
done
if [ -z "$fic" ]
then print -u2 "$usage"
     exit 1
fi

# TRAITEMENT

print PROG $option1 $option2 $val2 $fic

exit 0
```

VI.5.2. Et voici plusieurs situations de lancement :

```
$ prog
./prog: usage: prog [-opt1] [-opt2 val] fichier
$ prog -opt1
./prog: usage: prog [-opt1] [-opt2 val] fichier
$ prog -opt2 tutu toto
PROG 0 1 tutu toto
$ prog -opt2 -opt1 tutu toto
./prog: usage: prog [-opt1] [-opt2 val] fichier
$ prog -opt2 tutu -opt1 toto
PROG 1 1 tutu toto
$ prog -opt2 tutu -opt1
./prog: usage: prog [-opt1] [-opt2 val] fichier
$ prog toto
PROG 0 0 toto
$ prog -tutu
./prog: (-tutu) option inconnue
./prog: usage: prog [-opt1] [-opt2 val] fichier
```

Sachez qu'il existe aussi une commande UNIX qui permet de tester la syntaxe des options : `getopt`. On passe en paramètre à cette commande une chaîne contenant la liste des options autorisées, et leur type (option suivie ou non d'une valeur associée), en plus des paramètres de position. La commande vérifie que la syntaxe est correcte. Mais le nom de l'option ne peut dépasser un seul caractère (cela donne des options du style : `-a`, `-n` *2*, `-f` *toto*, etc...).

VI.5.2. Test de la validité des paramètres

Il faut s'assurer lorsqu'on utilise un paramètre qu'il correspond à la valeur qu'on attend ; ce n'est pas toujours facile à vérifier, mais très utile.

Imaginez que le premier paramètre à fournir à un programme soit un nombre et le second un nom de fichier (pour le résultat) : vous devez supposer qu'un utilisateur, voire vous-même, inverserez au moins une fois ces deux paramètres au moment de taper la commande.

Donc, il faut essayer de caractériser les différents paramètres et de les reconnaître ;

Exemples :

- le nombre de paramètres (y compris les options) doit être au moins égal à 4 :

  alors faire un test : `if [ $# -ge 4 ]`

- le paramètre $2 est un nom de fichier :

  alors faire un test : `if [ -f $2 ]`

- le paramètre $1 est un nom de fichier de données (on doit donc pouvoir le lire) :

VI.5.2.

alors faire un test : `if [ -r $1 ]`

- le paramètre $1 est un nom de fichier qui ne doit pas être vide :

  alors faire un test : `if [ -s $1 ]`

- le paramètre $1 est un nom de fichier résultat (on doit donc pouvoir y écrire dedans) :

  alors faire un test : `if [ -w $1 ]`

- le paramètre $1 est un nombre entre 1 et 20

  alors faire un test : `if [ $1 -ge 1 ] && [ $1 -le 20 ]`

- le paramètre $3 est une chaîne commençant par toto et suivie par _un ou _deux

  alors faire un test : `if [[ $3 = toto@(_un|_deux) ]]`

- le paramètre $2 est une chaîne de 9 caractères :

  alors faire un test : `if [ ${#2} = 9 ]`

etc ...

VI.5.2.

VI.6. Entrée standard et/ou fichiers nommés

Vous avez remarqué qu'un grand nombre de commandes UNIX sont conçues pour que, si le nom du fichier d'entrée n'est pas présent parmi les paramètres, la commande attende les données sur l'entrée standard.

Exemple :

> `grep chaîne toto` comporte le nom du fichier de données comme paramètre de la commande, mais `grep chaîne <toto` ou `cat toto | grep chaîne` marchent aussi, et là, les données sont attendues sur l'entrée standard parce que le paramètre correspondant au nom du fichier de données n'a pas été fourni.

L'intérêt principal est qu'on peut, si on désire, "piper" le programme au milieu d'autres commandes UNIX.

Supposons que votre programme soit appelé avec un paramètre qui est le nom du fichier à traiter. Voici la façon de structurer le programme pour qu'il accepte aussi l'entrée standard si ce paramètre est absent :

```
#!/bin/ksh
#
# exemple de programme acceptant
# un nom de fichier comme parametre
# ou l'entree standard par defaut
#

cat $1 |
(
 while read ligne
 do
    # VOTRE TRAITEMENT
 done
)
```

1er cas : on appelle le programme ci-dessus en spécifiant un paramètre ; alors la variable $1 est instanciée, et la commande `cat` envoie dans le pipe le contenu du fichier spécifié.

2ème cas : on appelle le programme sans paramètre ; dans ce cas, la variable $1 n'est pas instanciée, et au moment de l'interprétation de la ligne, elle est remplacée par *rien*. Ce qui fait que la première ligne du programme est alors équivalente à :

```
cat  |
```

La commande `cat` n'ayant pas de paramètre, elle attend les données sur l'entrée standard, qui par héritage, est celle du programme.

Donc, on peut utiliser ce programme avec l'entrée standard, et par conséquent, le "piper".

L'exemple présente un traitement de fichier encapsulé dans une boucle `while-do-read-done` : il est possible de remplacer toute cette partie (encadrée par des ()) par une commande UNIX simple comme `awk`, `sed`, etc... VI.5.2.

C'est d'ailleurs préférable, pour des raisons de performances, dès que votre fichier d'entrée est d'une taille importante[1].

1. Voir le chapitre précédent, consacré à ce sujet.

VI.5.2.

VI.7. Manipulation des fichiers temporaires

Lorsque vous avez besoin d'un fichier temporaire, il faut toujours penser :

- qu'il devra être détruit lorsque le programme sera terminé ;
- que plusieurs programmes peuvent peut-être tourner ensemble, et qu'à chaque lancement un nom spécifique doit être trouvé.

Où mettre le fichier?

Dans un répertoire "fourre-tout", comme `/tmp`, ou `/usr/tmp` qui existent quelque soit la machine, et qui sont autorisés à l'écriture pour tous.

De plus, ces répertoires sont généralement vidés de temps en temps, et les fichiers temporaires perdus ne s'y accumuleront pas; mais il ne s'agit pas de compter la-dessus pour travailler "salement" !!

Comment le nommer?

Notez tout d'abord que dans certaines (anciennes) version d'Unix, ou par choix de l'administrateur, vous ne disposez que de 14 caractères. Partons sur cette base.

Chaque lancement de programme correspond à un processus de numéro unique sur une machine : il est donc pratique que le nom du fichier temporaire contienne le numéro de processus courant (sur 5 chiffres, disponible dans la variable $$) ;

Chaque fichier du programme peut être différencié par un caractère particulier (A pour le premier, B pour le second, etc...).

Il reste 8 caractères que vous pouvez affecter avec le nom du programme.

Exemple : votre programme se nomme calcul, et il utilise deux fichiers temporaires ; vous pouvez les définir de la manière suivante :

```
tmp1=/tmp/calculA$$
tmp2=/tmp/calculB$$
```

Une première exécution verra par exemple la création de deux fichiers :

`/tmp/calculA12481` et `/tmp/calculB12481`

alors qu'un deuxième lancement en parallèle verra alors (par exemple) la création de :

`/tmp/calculA12485` et `/tmp/calculB12485`

Dans votre programme, vous manipulez les fichiers temporaires non plus par leur nom réel, mais par l'intermédiaire des variables; le contenu de la variable est le nom réel du fichier (qui change à chaque appel du programme), alors que le nom de la variable est évidemment toujours le même, et peut être significatif du contenu du fichier correspondant.

Au lieu de faire, par exemple :

```
cp /tmp/calculA$$ /tmp/calculB$$
```

vous écrirez alors :

```
cp $tmp1 $tmp2
```

VI.8. Traitements des signaux

VI.5.2.

Il peut être intéressant de traiter les signaux pour deux raisons opposées

- on veut ignorer certains signaux : INT (l'équivalent du <control>C), HUP (signal de déconnexion)
- on veut réagir à certains signaux

Le premier cas se rencontre lorsqu'on veut ignorer des signaux qui interrompent le programme en cours d'exécution. Dans cette situation, on place en début de programme

`trap` "" *SIG1 SIG2 SIG3 ...*

pour éviter que les signaux *SIG1, SIG2, SIG3*, etc ne soient pris en compte.

Le deuxième cas se rencontre par exemple si on utilise des fichiers temporaires. Supposons que le programme est en cours d'exécution et qu'il a créé des fichiers temporaires : si on interrompt le programme par <control C>, le processus est tué, mais les fichiers temporaires déjà créés ne sont pas détruits.

Il est alors intéressant de dérouter le signal INT (provoqué par le <control C>) vers une fonction qui supprime les fichiers temporaires.

Exemple :

```
#!/bin/ksh
#
# structure de suppression des fichiers
# temporaires sur interruption.

# declaration des fichiers temporaires
tmp1=/tmp/testA$$
tmp2=/tmp/testB$$

# fonction d'arret
arret()
{
 echo "$0: arret."
 rm -f $tmp1 $tmp2
 exit 0
}
# deroutement des signaux
trap "arret" INT HUP TERM

# TRAITEMENT
sleep 1000
```

Dans le début du programme, on déclare successivement

- le nom des fichiers temporaires (ou des objets à traiter en cas d'interruption par un signal) ;

VI.5.2.

- la fonction qui sera lancée à la réception de l'interruption ;
- les déroutements à considérer par la commande interne `trap`.

Ici, tout signal INT, HUP ou TERM reçu après l'exécution de la commande `trap` sera dérouté vers la fonction `arret.`

VI.9. Emission des messages d'erreur

VI.5.2.

Si vous faites des tests de validité à l'intérieur de votre programme, vous serez vraisemblablement amenés à générer des messages d'erreur.

Pour respecter la convention UNIX, il est préférable d'écrire ces messages dans la sortie d'erreur :

`echo` "*message_erreur*" >&2

ou bien :

`print -u2` "*message_erreur*"

De plus, si votre programme ou utilitaire est intégré dans d'autres commandes, il est très utile durant l'exécution de connaître la provenance exacte du message d'erreur, ceux-ci étant souvent ambigus hors du contexte.

Donc, il est intéressant de faire précéder le message d'erreur du nom du programme qui l'a produit. Exemple :

`print -u2` "$0 *: message_erreur*"

la variable $0 étant le premier champ de la ligne de commande, c'est à dire le nom du programme lui-même.

Il se peut que $0 contienne le chemin absolu du programme (s'il a été lancé en spécifiant le chemin absolu), ce qui n'est pas très joli. Plutôt que $0, on peut alors utiliser `${0##*/}`

VI.5.2.

VI.10. Génération du code de retour

Si vous ne générez pas de code de retour à la fin de votre programme, celui-ci est égal au code de retour de la dernière commande UNIX exécutée.

Un code de retour est utile car lorsque votre programme est encapsulé[1], c'est un moyen pratique de tester s'il s'est terminé correctement ou non.

On peut forcer le code de retour par la commande `exit` *n*. Cette commande provoque l'arrêt de l'exécution et génère un code de retour de valeur *n*.

La convention veut qu'une commande qui s'est bien terminée renvoie un code de retour nul.

1. c'est à dire lancé à partir d'une autre commande, UNIX ou non.

VI.5.2.

VII. Quelques messages d’erreur courants

VII.1. Format du message

VI.5.2.

La plupart des messages d’erreur apparaissent sous la forme suivante :

fichier: *localisation*: *message d’erreur*

avec :

- *fichier* : le nom de la commande ou du fichier lancé qui détecte l’erreur.

Cela peut être :

- ksh si vous étiez en train de taper une commande dans une fenêtre : votre commande est inconnue ou impossible à lancer :

```
$ gros_gag
ksh: gros_gag: not found
$
```

=> ici, l’interpréteur n’arrive pas à lancer la commande gros_gag

- un nom de commande UNIX : la commande a démarré, mais par suite d’un mauvais paramètre, de ressources ou de droits insuffisants, (etc...), elle s’interrompt et vous donne le message d’erreur :

```
$ grep toto titi
grep: can’t open titi
$
```

=> ici, la commande grep n’arrive pas à accéder au fichier titi.

- un nom de fichier binaire exécutable crée par vous : idem cas précédent,

- ou alors le nom de votre script shell, qui ne respecte pas la syntaxe, ou a rencontré une commande impossible à lancer. Dans ce cas, *fichier* est suivi par une séquence : [<*numéro*>] qui indique que l’erreur s’est produite pendant l’interprétation de la ligne <*numéro*> [1] :

```
$ gros_bug
./gros_bug[2]: ppppp : not found
./gros_bug[2]: syntax error at line 4: ’then’ unmatched
$
```

voir voir

=> ici, le script gros_bug contient une commande inconnue (ppppp) et une erreur de syntaxe.

1. Le script doit démarrer par : #!/bin/ksh sinon, les numéros ne sont pas fournis.
Attention, la notation est un peu bizarre : on note les lignes à partir du haut du fichier, en sautant celles qui ne peuvent être interprétées ; par exemple, si on a deux messages d’erreur qui se suivent, le numéro de ligne indiqué dans le deuxième ne tient pas compte de la ligne erronée qui précède : il y a décalage de 1 entre le numéro de ligne indiqué et le véritable dans le fichier script.

VI.5.2.

- *localisation* :

Apparait lorsque *fichier* est l’interpréteur de commande (ksh) ou un fichier script. Si *fichier* est un binaire exécutable, cette zone n’est pas renseignée.

```
$ more gros_bug
#!/bin/ksh
ppppp
if [ -f gros_bug ]
then echo "gros_bug existe"
date
$ gros_bug
./gros_bug[2]: ppppp: not found
./gros_bug[2]: syntax error at line 4: 'then' unmatched
$
```

Cela peut être :

- un nom : celui d’une commande (ppppp) trouvée pendant l’exécution du script(gros_bug), et que le shell n’arrive pas à lancer ;

- le message "syntax error at line .."avec le numéro de ligne du script *fichier* (gros_bug) à partir de laquelle le shell détecte un erreur de syntaxe.

Dans ce cas, le shell était en train d’interpréter un ensemble de lignes, qui commence à <*numéro*> (2), et finit au numéro indiqué par "syntax error at line .."(4) : il vous faut chercher votre erreur entre ces deux lignes[1]

- *message d’erreur* :

le message en clair (!?) qui correspond à l’erreur détectée.

(voir ci-après)

1. ici, la ligne 4 correspond à la fin de fichier, et le shell a détecté que la séquence if qui démarre à la ligne 2 n’a pas de fin.

VI.5.2.

VII.2. Liste des principaux messages d'erreur

et leur signification (la plus courante) :

`arg list too long`

la ligne de commande est trop longue une fois développée.

`bad file unit number`

on essaie d'utiliser un numéro de descripteur de fichier qui n'a jamais été affecté. (voir les redirections)

`bad number`

le paramètre numérique passé en argument est incorrect (trop grand, trop petit, ce n'est pas un nombre, etc...)

`bad option(s)`

on essaie de lancer un ksh avec une mauvaise option.

`bad substitution`

un opérateur sur une variable (`${ }`) défini de manière incorrecte.

`bad trap`

le numéro de signal spécifié pour la commande `trap` est incorrect.

`Bus error`

l'exécutable binaire lancé a fait une erreur d'adresse.

=> bug dans le programme.

`cannot create`

les droits en écriture sur le répertoire dans lequel on veut créer un fichier ne sont pas suffisants.

`cannot dup`

impossible de faire une redirection (probablement, trop de fichiers ouverts à la fois dans le programme)

`cannot execute`

droits en exécution insuffisants ou entête de script incorrect

`cannot fork: no swap space`

impossible de créer un nouveau processus (de lancer une commande) : pas assez de swap

`cannot stat .`

le répertoire courant a été supprimé (par un autre processus) et n'existe plus au moment où vous tapez la commande `pwd`.

`cannot open`

droits en lecture insuffisants sur le fichier ou le répertoire spécifié

VI.5.2.

`core dumped`

voir voir

suite à une erreur grave (erreur d'adresse, de bus, etc...) une commande a été interrompue, et le système recopie une image de la mémoire dans un fichier nommé `core`, en vue d'une éventuelle investigation.

Dans 99,99% des cas, l' "éventuel" se trouve être "improbable", et comme un fichier core est assez, voire très gros, il est bon de penser à le détruire.

`end of file unexpected at line ...`

dans un programme shell, une séquence comportant un caractère (ou un terme) de début puis de fin (par exemple : **`" "`** , ou **`if fi`**, ou **`do done`**, etc..) est incomplète : le terme final est manquant.

`Executable file incompatible with hardware`

le binaire que vous essayez de lancer n'a pas été produit sur une machine ayant une architecture comparable.

`execute permission denied`

droits en exécution insuffisants sur le fichier spécifié

`Floating exception`

division par zéro (problème de calcul)

`fork failed - too many processes`

nombre de processus sur la machine ayant atteint son maximum

`Hangup`

la connexion a été perdue (problème réseau, fermeture de la fenêtre dans laquelle la connexion a été lancée, etc...) ou alors réception du signal HUP (numéro 1)

`is not an identifier`

la commande shell a été lancée avec un nom de variable incorrect (ou alors un caractère non attendu est placé dans le nom de variable).

ce message apparait aussi si une commande :

`export` *`var=valeur`*

est fournie à un Bourne Shell (`/bin/sh`) : si les noms de variables sont corrects, cela peut indiquer que votre programme n'a pas été lancé avec un KornShell, mais avec un Bourne Shell ;(vérifier la présence du `#!/bin/ksh` en tout début de fichier).

urne Shell (`/bin/sh`) : si les noms de variables sont corrects, cela peut indiquer que votre programme n'a pas été lancé avec un KornShell, mais avec un Bourne Shell ;(vérifier la présence du `#!/bin/ksh` en tout début de fichier).

`is read only`

droits en écriture insuffisants sur le fichier spécifié

`Killed`

le processus a reçu un message KILL (numéro 9)

VI.5.2.

`ln: different file system`

la commande `ln` tente de créer un lien physique entre deux fichiers qui se trouveraient alors sur deux unités de disques différentes : impossible.

Si possible, utiliser un lien symbolique (avec `ln -s`)

`Memory fault`

l'exécutable binaire lancé a fait une erreur d'adresse.

=> bug dans le programme.

`more tokens expected`

probablement que dans un test, vous n'avez pas encadré les arguments entre double-quotes.

`mv: can't mv directories across file systems`

la commande `mv` est utilisée pour déplacer un répertoire d'une position à une autre n'appartenant pas à la même unité de disque : impossible.

Copier le répertoire à déplacer avec un `cpio` par exemple (voir les cas types), et supprimer l'ancien.

`no space left on device`

plus de place sur le disque.

`not found`

le fichier est inexistant, le chemin indiqué pour le fichier est incorrect, ou alors le répertoire contenant la commande lancée n'est pas dans le PATH[1]

une autre possibilité est que l'un des répertoires du chemin spécifié n'ait pas les droits en exécution.

`no match`

dans le répertoire spécifié ou courant, aucun fichier n'a de nom correspondant à l'expression indiquée (avec des caractères spéciaux comme *, [], ?,etc...)

`No such file or directory`

le nom fourni est mauvais, ou alors il correspond à un lien symbolique pointant sur un fichier ou répertoire inexistant.

`parameter not set`

la variable spécifiée n'a pas été affectée préalablement et le shell est en "mode" `-u` (voir la commande `set` du shell)

`Permission denied`

le fichier que vous voulez écraser, le répertoire dans lequel il se trouve si vous essayez de le déplacer, ne vous appartiennent pas et n'ont pas les droits suffisants.

`Terminated`

processus interrompu par un signal TERM (numéro 15)

1. il doit y avoir **.** dans le PATH pour que les commandes placées dans le répertoire courant soient exécutées.

VI.5.2.

`test argument expected`

il manque un argument entre les crochets d'un `if` ou d'un `while`

le plus souvent, cela correspond à une ligne :

```
if [ $ma_variable = "valeur" ]
```

avec *ma_variable* qui n'est pas affectée ou qui vaut une chaîne vide. Encadrer *$ma_variable* avec des double-quotes (")

`test ] missing`

il manque un crochet au `if` ou au `while`

`text busy`

le fichier que vous voulez modifier, supprimer, ou déplacer est un binaire en cours d'exécution ; il faut attendre sa fin pour pouvoir y toucher.

`too big`

plus assez de place mémoire pour pouvoir lancer le programme

`unknown test operator`

le symbole utilisé pour le test est incorrect (dans le `if [ ... ]` ou le `while [ ... ]`); plus probablement, vous n'avez pas encadré les arguments entre double-quotes, et le shell prend une partie de votre premier argument pour l'opérateur.

VIII.

VIII. Spécificités SUN-OS 5.1

Il est rappelé que sont ici indiquées des différences par rapport aux explications ou exemples fournis dans la description qui précède (sous HP-UX), sans que l'une ou l'autre des versions ne constitue une quelconque référence ou norme.

Vous trouverez les modifications par ordre d'apparition dans le document.

• Voir page 32:

Il est n’est pas possible de changer les propriétés du lien une fois celui-ci créé : chaque accès au lien est transparent, et concerne en fait le fichier sur lequel le lien pointe.

• Voir page 49;

Les messages d’erreur diffèrent un peu d’une version d’OS à une autre; ici, on a en fait:

```
$ ls toto tata | wc -l
toto: No such file or directory
tata: No such file or directory
 0
$ ls toto tat 2>&1 | wc -l
 2
```

• Voir page 53:

Le contenu du répertoire `/usr/pub` est différent.

• Voir page 57

Les fenêtres générées par défaut varient en fonction de l’OS; dans le cas de SUN-OS, qui est livré avec OpenWindow,ce sont des `cmdtool`.

Les caractères de contrôle étant différent, voici l’implémentation de la fonction `cdx` sur une `cmdtool` :

```
#
# affichage sur une fenetre cmdtool du repertoire et
# de la machine courante
#
PATH=/bin:/sbin:/usr/bin:/usr/sbin:/etc:/usr/ucb
EDITOR=vi
export HOSTNAME=$(hostname)
typeset -L4 NHOST=$HOSTNAME
export USER=$(whoami)

xname()
{
 _title="$*"
 print '\033]l'"$_title"'\033\' '\033]L'"$_title"'\033\' "\b\b\c"
}
cdx()
{
 unalias cd
 cd $1;
 alias cd=cdx
 xname $HOSTNAME [pwd=$PWD]
}
alias cd=cdx
set +u
cd
```

- Voir page 60 :

  L'intitulé des données affichées change un peu :

  - la dernière colonne est notée : `COMD`.

  La signification ne change pas.

- Voir page 64 :

  la commande `suspend`, c'est à dire `kill -STOP` est valide, mais réagit avec un temps de retard: comme on peut le voir dans l'exemple qui suit, l'indication de changement d'état ne s'effectue que lorsque une commande UNIX (`jobs` est une commande interne du shell) est lancée:

```
$ alias suspend='kill -STOP'
$ sleep 1000 &
[1] 25039
$ jobs
[1] + Running sleep 1000 &
$ suspend %1
$ jobs
[1] + Running sleep 1000 &
$ ls
des jef nouveau prof_p
[1] + Stopped (signal) sleep 1000 &
$ jobs
[1] + Stopped (signal) sleep 1000 &
$
```

Par contre, le signal a bien été pris en compte de suite: on peut s'en assurer en faisant le même essai que ci-dessus et en remplaçant le `sleep 1000` par :

```
sh -c "sleep 10; echo toto >/tmp/toto"
```

Il suffit depuis une autre fenêtre de surveiller l'exitence du fichier `/tmp/toto;` si, une fois exécuté la commande `suspend` , le fichier apparait, c'est donc que le process `sleep` a continué à s'exécuter, et que le fichier `/tmp/toto` a été créé; en pratique, rien ne se passe: le signal `STOP` a bien été reçu.

• Voir page 65 :

La liste des signaux est identique si on s'en tient aux 19 premiers; pour les autres, il y a des différences:

```
$ $ kill -l
 1) HUP          12) SYS          23) STOP
 2) INT          13) PIPE         24) TSTP
 3) QUIT         14) ALRM         25) CONT
 4) ILL          15) TERM         26) TTIN
 5) TRAP         16) USR1         27) TTOU
 6) IOT          17) USR2         28) VTALRM
 7) EMT          18) CHLD         29) PROF
 8) FPE          19) PWR          30) XCPU
 9) KILL         20) WINCH        31) XFSZ
10) BUS          21) URG          32) WAITING
11) SEGV         22) POLL         33) LWP
$
```

Pour tous ceux dont l'utilité a été indiquée, la signification est la même.

• Voir page 76 :

L'option `-E` pour la commande `grep` n'existe pas; on peut remplacer cependant le `grep -E` par la commande équivalente `egrep`.

Toutefois, certaines expressions régulières ne sont pas totalement supportées, en particulier la séquence `ER\{n,m\}` (elle permet d'indiquer un nombre de répétition donné pour une expression `ER`) qui n'existe pas. Cette remarque s'applique de la même façon à toute les séquences contenant des `{}`, aucune n'étant supportée.

• Voir page 78 :

Cette séquence n'est pas disponible. Voir ci-dessus.

• Voir page 79:

Les exemples `([abc]z)\{3\}` , ou `([abc])\{3\}` ne sont pas reproductibles, la séquence \{ \} n'étant pas disponible. Voir ci-dessus.

• Voir page 94:

La commande `what` n'est pas dans le PATH de défaut: il faut indiquer sont chemin complet: `/usr/ccs/bin/what`; de plus le résultat de la commande est différent, car il contient la liste de tous les modules constituant la commande.

En outre, il faut savoir que la vieille version du `awk` est fournie encore en standard sous SOLARIS 5.1. Si vous voulez utiliser les fonctionnalités avancées indiquées dans ce document, vous devez utiliser la commande `nawk` (contraction de *new awk*)en lieu et place de `awk`.

Dans la suite du document, les indications données pour `awk` devront être comprises avec SOLARIS 5.1 comme se rapportant à `nawk`.

- Voir page 109

La commande `bs` n'est pas disponible. Si on se contente de calculs sur des entiers, on peut utiliser `bc`, qui possède aussi des fonctions arithmétiques évoluées (racine, exponentielle, etc..).

- Voir page 112:

La liste est moins complète, mais les types principaux sont reconnus.

- Voir page 115:

L'option `-q` de la commande `grep` n'existe pas; ici, il est possible d'utiliser l'option `-l`, qui provoque l'affichage du fichier contenant la chaine spécifiée; et comme la commande `grep` affiche le nom du fichier, le `find` peut éviter de le faire.

La commande complète devient:

```
find . -exec grep -l chaine {}\; ! -exec rm {} \;
```

- Voir page 116:

Idem ci-dessus; l'option n'est pas disponible.

- Voir page 117:

Les options pour indiquer les colonnes de jointage sont différentes: au lieu de -1 et -2, il faut utiliser -j1 et -j2.

```
$ join -j1 2 -j2 1 -o 1.2 2.2 1.3 1.1 j_tab1
j_tab2
Cabours 62 Paimpol 205
Dupond 53 Paris R4
Durand 22 Nice ferrari
Pierre 35 Lyon 405
$
```

- Voir page 118:

L'option -v n'existe pas: on ne peut envisager d'utiliser que l'option -a qui rajoute les lignes non appariées à la sortie standard.

- Voir page 129 :

L'option -k n'existe pas: utiliser l'ancienne notation.

• Voir page 129 (Exemple simple) :

Voir ci-dessus; la commande à taper est:

```
sort +1 sort.input
```

• Voir page 130 (remarques) :

Voir ci-dessus; la commande à taper est:

```
sort +1 -2 sort.input
```

• Voir page 130 (séparateurs multiples) :

Voir ci-dessus; la commande à taper est:

```
sort -b +1 sort.input
```

• Voir page 130 (tri multiple) :

Voir ci-dessus; la commande à taper est:

```
sort -b +1 -2 +0 -1 sort.input
```

• Voir page 131 (exemple complexe) :

Voir ci-dessus; la commande à taper est:

```
sort +rn1 +0 sort.input
```

• Voir page 135 :

La commande `rev` n'est pas disponible.

• Voir page 158 :

La séquence pour étendre le nom du fichier en cours de frappe est :

`<escape>` ****

au lieu de `<escape>` `<escape>`

• Voir page 159 :

dem ci-dessus: chaque séquence de deux `<escape>` doit être remplacée par une séquence `<escape>` suivi de ****

• Voir page 160 :

Lorsque la commande cmd_bg & est lancée, le message "Stopped(tty output)" n'apparait pas tout de suite, même si on tape plusieurs <Return>; il apparait seulement après qu'une commande UNIX ait été tapée (une commande Shell n'est pas suffisante).

Malgré cette différence, les fonctionnalités ne sont en rien modifiées.

```
$ cmd_bg &
[1] 4518
$
$
$ jobs
[1] + Running cmd_bg &
$
$
$ id
uid=17000(reseau) gid=17000(reseau)
[1] + Stopped(tty output) cmd_bg &
$
```

• Voir page 166 :

se référer aux remarques sur la commande `sort` vues plus haut. Avec l'ancienne notation, la commande est inchangée, par contre, la nouvelle notation n'est pas disponible.

• Voir page 166 :

idem ci-dessus. La nouvelle notation n'étant pas disponible, la commande devient:

`sort +2 -4 +0` *`mon_fichier`*

• Voir page 173 :

la commande `remsh` se nomme `rsh` sur Sun.

• Voir page 174 :

l'option -linkedto n'est pas disponible; il faut donc rechercher les fichiers à partir des numéros d'inode, comme cela est expliqué dans le deuxième exemple.

• Voir page 177 :

la commande se trouve dans le répertoire `/usr/ccs/bin`, qui n'est généralement pas intégrée dans le PATH: il faut alors taper le chemin complet.

• Voir page 178 :

il n'y a pas d'entrée correspondant au terme `suffix`.

• Voir page 185 :

la commande `bs` n'est pas disponible.

• Voir page 206 :

la remarque en bas de page ne prend pas effet.

• Voir page 209 :

On peut rajouter :

`couldn't set locale correctly`

la variable LANG n'est pas positionnée correctement. (langage courant du pays). Une valeur par défaut passe partout est : **C**.

IX.

IX. Spécificités AIX 3.2

Il est rappelé que sont ici indiquées des différences par rapport aux explications ou exemples fournis dans la description qui précède (sous HP-UX), sans que l'une ou l'autre des versions ne constitue une quelconque référence ou norme.

Vous trouverez les modifications par ordre d'apparition dans le document.

• Voir page 31 :

Les messages d'erreur étant dispensés en français, on a en fait:

```
$ ln ancien /tmp/nouveau
ln: 0653-422 ancien et /tmp/nouveau se trouvent sur des systmes de fichiers diffrents.
$
```

Pour un utilisateur averti d'Unix, les messages traduits en français sont parfois déroutants; pour un néophite, ils sont, comme ici, généralement plus accessibles.

• Voir page 32 :

Il est n'est pas possible de changer les propriétés du lien une fois celui-ci créé : chaque accès au lien est transparent, et concerne en fait le fichier sur lequel le lien pointe.

• Voir page 53 :

Le contenu du répertoire `/usr/pub` est différent.

• Voir page 57 :

Les fenêtres générées par défaut varient en fonction de l'OS; dans le cas de AIX,,ce sont des `aixterm.`

Les caractères de contrôle étant différent, voici l'implémentation de la fonction `cdx` sur une `aixterm`:

```
#
# affichage sur une fenetre aixterm du repertoire et
# de la machine courante
#
PATH=/bin:/sbin:/usr/bin:/usr/sbin:/etc:/usr/ucb
EDITOR=vi
export HOSTNAME=$(hostname)
typeset -L4 NHOST=$HOSTNAME
export USER=$(whoami)

xname()
{
 _title="$*"
 print '\033]l'"$_title"'\033' '\033]L'"$_title"'\033' "\b\b\c"
}
cdx()
{
 unalias cd
 cd $1;
 alias cd=cdx
 xname $HOSTNAME [pwd=$PWD]
}
alias cd=cdx
set +u
cd
```

• Voir page 60 :

L'intitulé des données affichées change un peu :

- la première colonne est notée : `USER`,
- la dernière `CMD`.

La signification ne change pas.

• Voir page 63 :

L'identificateur de processus courant est indiqué, en plus du descripteur de processus, lorsqu'un process change d'état.

```
$ sleep 1000
^Z[1] + 16417 Stopped sleep 1000
$ sleep 2000 &
[2] 30754
$ jobs
[2] + Running sleep 2000 &
[1] - Stopped sleep 1000
$
```

• Voir page 65 :

La liste des signaux est plus importante que pour les autres systèmes; cependant, à part quelques signaux, les numéros sont spécifiques.

Pour les principaux, dont l'utilité a été indiquée, la signification ne change pas.

```
$ kill -l
 1) HUP      17) STOP      33) DANGER    49) bad trap
 2) INT      18) TSTP      34) VTALRM    50) bad trap
 3) QUIT     19) CONT      35) MIGRATE   51) bad trap
 4) ILL      20) CHLD      36) PRE       52) bad trap
 5) TRAP     21) TTIN      37) bad trap  53) bad trap
 6) LOST     22) TTOU      38) bad trap  54) bad trap
 7) EMT      23) IO        39) bad trap  55) bad trap
 8) FPE      24) XCPU      40) bad trap  56) bad trap
 9) KILL     25) XFSZ      41) bad trap  57) bad trap
10) BUS      26) bad trap  42) bad trap  58) bad trap
11) SEGV     27) MSG       43) bad trap  59) bad trap
12) SYS      28) WINCH     44) bad trap  60) GRANT
13) PIPE     29) PWR       45) bad trap  61) RETRACT
14) ALRM     30) USR1      46) bad trap  62) SOUND
15) TERM     31) USR2      47) bad trap  63) SAK
16) URG      32) PROF      48) bad trap
$
```

• Voir page 66 :

L'affichage du changement d'état des process n'est pas instantané: il faut taper une commande pour que le message s'affiche:

```
$ sleep 1000 &
[1] 30763
$ sleep 2000 &
[2] 16428
$ jobs
[2] + Running sleep 2000 &
[1] - Running sleep 1000 &
$ kill -9 %1
$ kill -INT 16428
[1] - 30763 Killed sleep 1000 &
$ jobs
[2] + Interrupt sleep 2000 &
$
```

• Voir page 76 :

Le manuel ne fournit aucune indication se rapportant au terme `regexp`.

De plus, certaines expressions régulières ne sont pas supportées exactement de la même manière, en particulier la séquence `ER\{n,m\}` (qui permet d'indiquer un nombre de répétition donné pour une expression `ER`): elle doit être notée sans les anti-quotes: **`ER{n,m}`**. Cette remarque s'applique à toute les séquences contenant des `{}`.

• Voir page 78 :

Ce type d'expression est disponible, mais il ne faut pas mettre de **`\`** devant les **`{`** ou les **`}`**. Voir ci-dessus.

• Voir page 94 :

le résultat de la commande est différent, car il contient la liste de tous les modules constituant la commande.

• Voir page 96 :

Il n'est pas possible d'indiquer un séparateur comportant plus d'un caractère: l'exemple n'est pas reproductible.

• Voir page 100 :

En théorie, la séquence `%X` génère une conversion d'un décimal en hexa avec des lettres majuscules, alors que `%x` fait une conversion héxa en minuscules. Ici, la séquence `%X` (majuscules) n'est pas supportée; mais `%x` est utilisable.

• Voir page 112 :

La liste est moins complète, mais les types principaux sont reconnus.

• Voir page 129

Lorsqu'on utilise la nouvelle notation (avec -k), toutes les positions démarrent à un, aussi bien pour les champs que pour les caractères.

L'ancienne notation positionne cependant toujours le premier champ ou le premier caractère à zéro.

Dans la suite du document, toutes les commandes utilisant la commande `sort` avec l'option -k devront être modifiées en augmentant systématiquement de un les indices de position des champs et des caractères.

Exemple: une clé 0.2,2.5 avec l'ancienne notation devient 1.3,3.6 avec la nouvelle (option -k).

• Voir page 129 (Exemple simple) :

Voir ci-dessus; la commande à taper est:

```
sort -k2, sort.input
```

• Voir page 130 (remarques) :

Voir ci-dessus; la commande à taper est:

```
sort -k2,3 sort.input
```

• Voir page 130 (séparateurs multiples) :

Voir ci-dessus; la commande à taper est:

```
sort -kb2, sort.input
```

• Voir page 130 (tri multiple) :

Voir ci-dessus; la commande à taper est:

```
sort -b -k2,3 -k1,2 sort.input
```

• Voir page 131(exemple complexe) :

Voir ci-dessus; la commande à taper est:

```
sort -krn2, -k1,2 sort.input
```

• Voir page 135 :

Voir ci-dessus; la commande à taper est :

```
sort -k3, tst_sort1
```

• Voir page 158 :

La séquence pour étendre le nom du fichier en cours de frappe est :

```
<escape>  \
```

au lieu de `<escape>` `<escape>`

• Voir page 159 :

idem ci-dessus: chaque séquence de deux `<escape>` doit être remplacée par une séquence `<escape>` suivi de ****

• Voir page 166 :

se référer aux remarques sur la commande `sort` vues plus haut. Avec l'ancienne notation, la commande est inchangée, par contre, avec la nouvelle, la commande devient:

`sort -k3` *`mon_fichier`*

• Voir page 166 :

idem ci-dessus. La commande devient:

`sort -k3,5 -k1,` *`mon_fichier`*

• Voir page 174 :

les options `-linkedto` et `-follow` ne sont pas disponibles. Il est donc possible de rechercher seulement les fichiers liés physiquement, en utilisant le numéro d'inode.

• Voir page 178 :

il n'y a pas d'entrée correspondant aux termes `regexp`, ou `suffix.`

• Voir page 206 :

la remarque de bas de page ne prend pas effet.

• Voir page 209 :

On peut rajouter :

`couldn't set locale correctly`

la variable LANG n'est pas positionnée correctement. (langage courant du pays). Une valeur par défaut passe partout est : **C**.

INDEX

Symboles

A

B

C

©jf pujol, 18 août 1993

D

E

Q

R

S

Bibliographie

Ouvrages de référence :

liste des commandes Unix (utilisateurs):

HP-UX Reference Section 1
HP-UX 8.05 Manual

formats type (fichiers, signaux, expressions régulières, etc...):

HP-UX Reference Section 5
HP-UX 8.05 Manual

KornShell:

Le KornShell, langage de commande et de programmation
M.I. BOLSKY, D.G. KORN
MASSON PRENTICE HALL

vi, ex:

Ultimate guide to the VI and EX text editors
Hewlet Packard
BENJAMIN/CUMMINGS

awk, ed, sed:

Text Processing: User's Guide
HP-UX 8.05 Manual

bc, bs, dc:

Number Processing: User's Guide
HP-UX 8.05 Manual

Pour compléter ses connaissances:

guide du C shell:

The UNIX C Shell Field Guide
Gail Anderson, Paul Anderson
PRENTICE HALL

guide du Bourne Shell (sh), du Csh, utilitaires du langage C, appels système:

Advanced Programmer's Guide to UNIX System V
Rebecca Thomas, PhD, Lawrence R.Rogers, Jean L.Yates
Osborne McGRAW-HILL