SYSTEME UNIX

Table des matières

Petit guide d'UNIX à l'usage des Martiens

Préface

En temps qu'enseignant d'IUT informatique, je suis amené à faire découvrir aux futurs informaticiens de notre nation les rudiments de la ligne de commande au travers d'un cours appelé « UNIX : système utilisateur ». Ce cours est l'occasion d'une rupture entre le monde convivial mais peu performant de l'environnement graphique tel que tout utilisateur le connaît et le monde obscure mais tellement inintéressant de l'informaticien où la ligne de commande est reine et l'ordinateur enfin dompté. C'est en quelque sorte un premier pas vers ce qui sera l'environnement futur de l'informaticien.

Durant ces années d'enseignement, j'ai parfois eu du mal à convaincre de jeunes étudiants que la ligne de commande était un point de passage nécessaire. Je n'y suis pas toujours arrivé et je n'y arriverai sans doute pas plus à l'avenir, mais le temps et l'expérience travaillent sans doute pour moi.

Pourquoi écrire ce petit ouvrage ? Simplement car mes paroles en cours s'envolent et la prise de note semble un art en voie de disparition. Alors peut être que les écrits resteront. Et si vous me demandez en quoi est-il important que les écrits restent, je vous répondrai simplement en prenant ma seconde casquette : celle d'un professionnel du secteur privé qui dans son activité croise chaque jour le résultat du travail de ces jeunes maintenant diplômés. Sous cette casquette je constate avec quelle difficulté ils ont du mal, trois ou quatre ans après, à écrire un simple script. Je vais illustrer cela par un exemple qui me tient à coeur.

Je travaille dans un domaine que l'on appelle la « Supply Chain » : il s'agit de déplacer les produits fabriqués par les usines dans les bons magasins de sorte à ce qu'ils soient disponibles au bon endroit et au bon moment pour nos clients. Vous l'aurez compris, ce genre de programme n'est pas de ceux qui s'installent sur un simple ordinateur personnel. Ils ne sont pas non plus constitués d'un unique programme autonome. Il s'agit en fait d'une constellation de programmes composites, basés sur des progiciels et de multiples technologies. Pour vous donner une idée plus précise, un tel programme demande environ 100 jours pour être installé sur un serveur... Ce type de programme, finalement très banal en entreprise, est constitué de nombreux scripts et fonctionne sur un environnement UNIX.

Pour en revenir à mon illustration, au sein de cette application demeure un script qui m'a particulièrement interpelé, ce script exécute un traitement très simple : s'assurer que la dernière ligne d'un fichier texte est bien « fin fichier ». Il permet alors de vérifier que le fichier est bien arrivé dans son intégralité avant de le passer au programme suivant qui le chargera ligne par ligne dans la base de données. Lorsque j'ai eu a faire pour la première fois à ce script, j'ai découvert, non seulement qu'il faisait pas loin de 200 lignes mais qu'en plus il ne fonctionnait pas en renvoyant toujours vrai, que le fichier se termine ou non par « fin fichier ». Il y a donc ici deux problèmes, le premier est bien sûr que le programme ne fonctionne pas et donc qu'il n'a pas été testé, mais en plus, un tel programme ne devrait demander qu'une seule et unique ligne de script pour répondre au cahier des charges...

Qu'est-ce que cela signifie ? D'un point de vue économique, l'écriture de ces 200 lignes de scripts ne fonctionnant pas a dû coûter à mon employeur quelque chose comme 500 €, mais le plus grave et le plus dur à estimer est l'impact que ce script a sur la production au jour le jour : nous pouvons considérer que les impacts de ce non fonctionnement ont coûté une demi-journée de travail et des retards dans l'exécution de l'application chaque fois qu'un fichier est arrivé incomplet. C'est donc au final plusieurs milliers d'euros que ce script aura coûté... quelques mois de salaire d'un

informaticien débutant et plein de cheveux en moins pour ceux qui subissent ce script. Car bien sûr un tel bug met un peu de temps à être corrigé car il ne se déclenche pas toujours et les problèmes n'apparaissent qu'en aval du programme, lorsque des données incomplètes sont chargées.

Cet exemple, un peu long, a pour but de rappeler aux lecteurs qu'en entreprise les bugs coûtent des mois de salaires, d'une part, et pour les inciter à se référer à cet ouvrage ou tout autre, s'ils étaient dans la situation, au cours de leur carrière, de devoir écrire un script shell alors qu'ils n'auraient plus trop en mémoire cette matière...

Chapitre 1 : La découverte du nouveau monde, UNIX

Unix a l'intérêt d'être un standard et de garantir ainsi un environnement de travail offrant des caractéristiques définies. Ces caractéristiques assureront la compatibilité de logiciels par exemple et permettront à l'utilisateur une certaine indépendance vis à vis de son fournisseurs. UNIX, outre la percée de Linux dans des environnements utilisateur est un système principalement à destination des serveurs ou des stations professionnelles dédiées aux calculs ou la CAO.

UNIX est ce que l'on appelle un Système d'Exploitation, tout comme peut l'être Windows. S'il n'existe qu'un seul Windows (aux versions près), il existe de multiples UNIX. UNIX est donc plus un standard qu'un système en tant que produit. On appelle les Systèmes d'Exploitation SE ou OS comme Operating System, le terme anglais étant plus couramment utilisé.

Un système d'exploitation est un logiciel dont le rôle est de simplifier l'accès au matériel pour les applications qui seront lancées par les utilisateurs. C'est ainsi qu'il gère les composants d'un ordinateur comme la mémoire, le stockage des fichiers, les périphériques son ou réseau. L'OS est donc un élément très proche et bien spécifique à un matériel donné. Nombre de fabriquants de matériel possèdent donc leur propre système qu'ils livrent avec leurs ordinateurs.

Dans ses autres missions, le système d'exploitation gère aussi les utilisateurs et ainsi l'ensemble des droits que peut avoir un utilisateur sur l'ordinateur. Il est ainsi possible de décider de répertoires que l'utilisateur ne pourra pas consulter ou d'interdire à certains l'usage de la carte son...

Un système d'exploitation offre aussi aux développeurs des bibliothèques de fonctions permettant par exemple la lecture et l'écriture de fichiers. Des exemples plus étendus couvrent par exemple l'utilisation de capacités graphiques avancées. C'est le cas de DirectX, un environnement de programmation intégré au système qui s'adapte aux cartes graphiques présentes dans votre PC pour en utiliser au mieux les capacités. Cette bibliothèque offre aux développeurs de jeux vidéo un ensemble de fonctions rendant la conception de jeux 3D bien plus simple et leur assure que leur développement fonctionnera quel que soit le matériel de l'utilisateur.

L'utilisation d'un système d'exploitation est une évolution majeure de l'informatique, au tout début de l'histoire de nos ordinateurs, ceux-ci n'en possédaient donc pas. Un ordinateur sans système d'exploitation peut être comparé aux premières consoles que certains ont peut-être connues... celles où rien ne se passe si vous n'insérez pas la cartouche adéquat contenant le jeu.
Une machine sans OS n'est constituée que d'éléments matériels et n'a donc rien à exécuter, l'utilisateur devra donc y insérer un programme pour que quelque chose se passe et seul ce programme fonctionnera. En outre, ce programme n'ayant pas d'OS devra par lui-même intégrer toutes les fonctionnalités d'accès au matériel, il ne fonctionnera, par conséquent, que sur un matériel 100% identique. Avantage toutefois de ce système, le programme 100% étudié pour une architecture donnée peut utiliser au mieux les capacités du matériel.
De très nombreux systèmes informatiques d'aujourd'hui tournent encore sans système d'exploitation, ils ne font toutefois pas partie des systèmes conventionnels tels que vous les

imaginez mais des systèmes embarqués, beaucoup plus nombreux. Ces systèmes sont basés sur des micro-contrôleurs (sortes de systèmes informatiques miniaturisés) et se trouvent dans les objets du quotidien comme les téléviseurs, voitures, plaques de cuisson...

Les systèmes d'exploitation n'apparaissent que vers la fin des années 60 alors que le besoin de partager un même ordinateur à plusieurs se fait sentir. En effet, les équipements de l'époque coutent quelques millions de dollars et la superficie leur étant nécessaire couvre quelques centaines de mètres carrés. Pas question, donc, d'en posséder plusieurs. Il est donc nécessaire de les partager.
Partager un ordinateur signifie être capable de lancer plusieurs programmes en même temps, par exemple, ce qui, nous l'avons vu, n'est pas possible sans un élément logiciel créant cette fonctionnalité. Partager un ordinateur signifie aussi gérer des utilisateurs, leur accorder des privilèges ou non. Ainsi sont nés les premiers systèmes d'exploitation, tout d'abord MULTICS puis UNICS dans les célèbres laboratoires d'AT&T (Bell Labs) puissante entreprise de recherche à l'époque. UNICS sera mono-utilisateur mais multi-tâches (il pourra exécuter plusieurs programmes en même temps). Il deviendra rapidement UNIX et évoluera jusqu'à une version 6 qui sera donnée aux universités et entreprises en 1977.

De ce don se créent plusieurs branches dont la première celle de Berkley (BSD) en 1978. Microsoft ne sera pas en reste en proposant XENIX en 1980, OS a la vie plutôt courte, disparaissant cinq ans plus tard. De nombreuses autres branches vont apparaître : chaque constructeur de matériel proposant sa version, adaptée à ses serveurs. Ainsi, HP propose HP-UX, IBM AIX, SUN SUNOS/SOLARIS, Silicon Graphics IRIX ...

En parallèle de ces UNIX apparaissent d'autres systèmes profitant des nouvelles niches offertes par l'informatique personnelle, niches devenant ce que vous en connaissez aujourd'hui. Les acteur de ce marché sont Microsoft et Mac. D'autre tenteront en vain de s'y installer : BeOs du Français Gasnier ou Next de Steve Jobs, échappé quelque temps de la firme à la pomme. Ce seront des échecs : le marché est tenu. Il n'est pas d'ordinateurs vendus sans système pré-installé issu de l'un des deux majeurs.

Le seul système qui parviendra à percer ce monopole, avant de convaincre et de s'étendre aux serveurs sera GNU/Linux. Il s'imposera par sa qualité et sa gratuité avant de séduire par ses fonctionnalités qui aujourd'hui n'ont plus grand chose à envier aux systèmes commerciaux. GUN/Linux n'est une branche d'aucun autre UNIX, il aura été développé de rien par son auteur Linus Torwalds en 1991 ; de là naîtront de multiples distributions : Red-Hat, OpenSuse... Une distribution est un assemblage spécifique de composants tous identiques.
Une distribution est un noyau (le coeur d'un OS) appelé ici Linux, associé à de nombreux programmes tels que le compilateur C, un environnement en ligne de commande et des outils divers qui sont eux issus de l'initiative GNU de la Free Software Fundation. C'est ainsi que les deux termes doivent être associés GNU/Linux. GNU travaille de son côté à la réalisation de son propre noyau : HURD.

En entreprise, les systèmes UNIX étant les plus nombreux et l'environnement de Windows n'ayant rien de très intéressant à nous apprendre, ce sont ces premiers que nous allons étudier dans les chapitres à venir.

Chapitre 2 : A la découverte de la ligne de commande

La ligne de commande, pour certains, semble être un bond dans les temps préhistoriques vis à vis des interfaces graphiques riches que connaissent les utilisateurs d'aujourd'hui. C'est cependant une vision totalement fausse de l'évolution informatique lorsque l'on prend en considération le fait que même Microsoft, sentant la baisse de ses ventes, prévoit dans ses versions serveurs un environnement en ligne de commande étendu.

L'environnement en ligne de commande n'est sans doute pas des plus agréables lorsqu'il s'agit de lancer des applications, copier des fichiers (bien que...) mais c'est en tout cas le seul où il soit possible d'automatiser et de répéter des opérations à réaliser régulièrement. Ce n'est donc sans doute pas l'outil idéal pour l'utilisateur mais bien l'outil le mieux adapté à l'informaticien, particulièrement dans des métiers tels que ceux de l'exploitation.

La ligne de commande ou plus précisément la programmation shell offre un environnement de développement très efficace lorsqu'il s'agit de manipuler des fichiers, autant pour les déplacer, renommer... que pour les traiter : trier, découper, rechercher...

Ce sont les possibilités de cet environnement que nous allons étudier au fur et à mesure des chapitres à venir, par l'utilisation des environnements de type UNIX.

Le Shell est un programme interagissant avec l'utilisateur, il possède ainsi une entrée qui est par défaut le clavier et une sortie par défaut : l'écran. L'utilisateur peut y saisir des commandes qui seront exécutées par le système. Il existe de nombreux shells. Celui qui sera couramment utilisé sous Linux est le BASH, le Shell SH est le shell le plus basique, il assurera la compatibilité des scripts que vous écrirez. Il existe de nombreux autres shell comme le KSH, CSH, TCSH ...
Un shell est un programme comme un autre, il est ainsi possible de lancer un shell en tapant son nom alors que vous en utilisez un autre.

Outre le lancement de commande, un shell intègre un langage de programmation avec la prise en compte de conditions et de boucles, la gestions de variables... tout comme le fait le C par exemple.

Les toutes premières commandes

Pour notre premier exemple nous allons réaliser le plus classique des programmes, le Hello World, consistant à afficher à l'écran cette simple phrase. La commande shell permettant d'afficher quelque chose est ***echo*** ainsi, pour réaliser cet affichage, nous taperons simplement la commande suivante :

paul@linux:/tmp/> ***echo Hello World***

Le résultat apparaîtra sur la ligne suivante.

Les commandes shell sont très nombreuses et chacune possède de multiples options. Pour en percer le mystère, il existe quelques commandes très utiles : le manuel ***man*** ou ***info*** qui nécessite de connaître le nom de la commande que l'on souhaite détailler. Les recherches par fonctionnalités se font en utilisant ***apropos***.

Ainsi pour tout savoir des options de la commande ***echo*** que nous venons de voir, il suffit d'exécuter la commande ***man echo***. Une page de manuel s'affichera alors. La touche ***q*** permet de sortir de l'éditeur.

La recherche de fonctions permettant la copie de fichiers pourra se faire par l'appel à la commande ***apropos "copier des fichiers"*** qui vous indiquera de consulter les commande ***cp*** ou ***install***.

Une dernière façon d'obtenir des informations sur l'usage d'une commande est d'utiliser l'aide intégrée à la commande elle même, souvent accessible par une option spécifique : ***--help***. Cette façon de faire n'est toutefois pas standardisée et ne fonctionnera donc pas toujours.

Une commande est généralement un programme qui va être exécuté pas le shell. Cette commande, comme nous l'avons vu prend des paramètres. Les paramètres sont séparés par des espaces. Ainsi ***echo Hello World*** correspond à l'exécution de la commande ***echo*** avec comme

premier paramètre ***Hello*** et comme second ***World***. Il est possible de réunir ces deux mots en un seul paramètre en les plaçant entre double quotte « " ». Ce qui pour cette commande ne change rien mais pour d'autres sera primordial.
Certains paramètres sont particuliers et appelés options. Les options sont généralement précédées d'un « ***-*** » ou d'un « ***--*** » . Les options changent le comportement du programme. Les options sont généralement facultatives.

La façon dont doit être utilisée une commande est décrite dans sa documentation, prenons l'exemple de ***echo*** et de ce que nous dit son manuel :

> NAME
> echo -- write arguments to the standard output
>
> SYNOPSIS
> echo [-n] [string ...]
>
> ***echo*** est donc une fonction écrivant la liste de ses arguments sur la sortie standard (comme nous l'avons vu). Cette fonction prend comme argument une liste de chaînes des caractères : indiquée par ***string*** et ***...*** qui nous indique que tous les arguments suivants seront considérés comme le précédent à savoir des chaînes de caractères à afficher. ***echo*** peut recevoir une option ici ***-n*** ; qui (et c'est indiqué plus loin dans le manuel) sert à ne pas ajouter de retour charriot en fin de ligne. Les arguments comme l'option sont notés entre « **[]** » ce qui signifie qu'il sont facultatifs. Il est donc possible d'appeler la commande ***echo,*** sans rien indiquer de plus, sans que cela provoque une erreur. Si les arguments ne sont pas entourés de crochets, ils sont alors obligatoires sans quoi la commande retournera une erreur.

Il est possible d'enchaîner plusieurs commandes en les séparant par un retour charriot ou par le caractère « **;** ». L'exécution est dans ce cas séquentielle – la commande suivante commencera après la fin de la précédente.
Les commandes peuvent êtres inscrites dans un fichier plutôt que d'être tapées dans la console, on parlera alors de scripts. Un script est un programme, exactement comme l'est un programme écrit en C, à la différence qu'un programme shell n'est pas compilé mais simplement interprété. En exemple, voici votre premier programme shell, le programme **helloWorld** :

```
#!/bin/sh
echo "Hello World !"
```

Ce contenu sera enregistré dans un fichier texte du nom de *helloWorld* et pourra être exécuté simplement de la façon suivante : ***sh ./helloWorld***
La première ligne du fichier indique quel shell doit être utilisé pour son exécution, les lignes suivantes représentent simplement un enchaînement de commandes comme vous les taperiez dans la console.
Nous verrons par la suite qu'il existe d'autres moyens d'exécuter ce même script et comment lui permettre d'utiliser des paramètres.

L'organisation des fichiers sous UNIX

Vous connaissez sans doute l'organisation des fichiers sous Windows, à savoir qu'un fichier est rangé dans un répertoire, lui même rangé dans un autre répertoire, lui même placé sur une partition particulière d'un disque dur ou tout autre support.
Il en va de même sur UNIX au détail près que la notion de partition de disque dur est considérée de façon différente. UNIX est organisé autour d'un répertoire racine appelé « / » (slash, racine ou root).

Tous les autres répertoires ou fichiers sont rattachés à celui-ci au travers d'une organisation arborescente classique. UNIX possède un certain nombre de répertoires que l'on peut considérer comme normalisés même s'il n'y a rien de très formel là-dessus. Ainsi nous allons trouver entre autre :

/

home/	-	la base des répertoires des utilisateurs
etc/	-	le répertoire de configuration
var/	-	le répertoire des logs
usr/	-	un des répertoires des programmes
tmp/	-	répertoire des fichiers temporaires

Unix intègre aussi quelques répertoires bien particuliers... Il faut comprendre que sur UNIX, tout est fichiers, ainsi, un périphérique tel que la carte son est un fichier un peu spécial qu'il est possible de lire (acquisition des données du micro) ou d'écrire (pour faire jouer à la carte le son souhaité). Par ailleurs, tout programme en cours d'exécution est lui aussi représenté par un fichier. On trouvera donc sur UNIX des répertoires contenant des fichiers un peu particuliers :

/

dev/	-	répertoire de périphériques du système
proc/	-	répertoire des programme en cours d'exécution.

Contrairement à Windows, UNIX ne gère pas la notion d'extension et il n'existe pas de notion historique de limitation à 8 caractères pour un nom de fichier, ainsi le type d'un fichier est déterminé par son contenu et non pas son nom. Il est donc possible d'appeler un fichier image : *image.jpg* comme sous Windows ou de l'appeler *image.jpg.doc,* comme *image.doc* ou simplement *image* sans que cela ne change quoi que ce soit.
Une commande spécifique permet de connaître le type d'un fichier d'après son contenu, il s'agit de la commande ***file*** prenant comme argument le nom d'un fichier. Ce qui peut donner par exemple :

paul@linux:/tmp/>***file monCours.doc***
monCours.doc: GIF image data, version 89a, 100x76

Il existe deux fichiers particuliers que l'on trouvera dans chaque répertoire :
- le premier est le répertoire « **..** » qui signifie répertoire père
- le second est « **.** » qui signifie répertoire courant

Ces fichiers servent à se déplacer dans l'arborescence par rapport à la position courante plutôt que par rapport à la racine. On parle alors de positionnement relatif.

Naviguer dans l'arborescence des fichiers

Se déplacer dans l'arborescence des fichiers UNIX est la base de ce qu'il faut connaître pour aller plus loin dans l'utilisation d'UNIX. Il y a donc pour cela un ensemble de commandes à connaître.
La première d'entre elle est la commande ***pwd*** : cette commande permet de se situer dans l'arborescence indiquant le chemin complet du répertoire courant par rapport à la racine.
Ce type de chemin, partant de la racine est appelé chemin absolu. Il est aussi possible d'indiquer un chemin, non pas en partant de la racine mais en partant du répertoire courant, dans ce cas on parlera de chemin relatif.

Revenons à ***pwd***, son usage est simple, il n'y a pas spécialement d'option à utiliser. La commande

affiche simplement le chemin du répertoire courant.

paul@linux:/tmp/>***pwd***
/tmp

Notez au passage que cette information est indiquée dans l'entête de la ligne de commande que l'on appelle aussi le prompt, ci-dessus « paul@linux:/tmp/> » Le prompt est une chaîne configurable indiquant que le système attend, de la part de l'utilisateur, une commande. Il est ici composé par le login de l'utilisateur « paul », le nom de l'ordinateur « linux » et le répertoire courant « /tmp/ ».

La commande permettant de changer de répertoire est ***cd*** pour ***c****hange* ***d****irectory*, elle prend en paramètre le chemin vers lequel vous souhaitez vous déplacer. Il est possible de lui indiquer un répertoire de façon absolu (en partant de la racine) ou relatif (en partant d'un répertoire qui peut être le répertoire courant ou celui d'un utilisateur).

Ainsi, où que l'on soit, il est possible de se rendre dans le répertoire de l'utilisateur *paul*, en indiquant un chemin absolu comme suit :

paul@linux:/tmp/>***cd /home/paul***
paul@linux:/home/paul/>

La commande ***cd*** n'affiche rien mais change le répertoire courant comme l'indique la modification du prompt.
Pour stipuler un déplacement à partir du répertoire d'un utilisateur, nous allons utiliser un symbole particulier : « ~ » appelé « tilde » sans rien de plus, ce symbole signifie « le répertoire de l'utilisateur courant », il est alors possible de lui adjoindre un nom d'utilisateur : « **~paul** » pour indiquer « le répertoire de l'utilisateur paul » ou « ~**root** » pour indiquer « le répertoire du super utilisateur root ».
Cette façon de faire permet de raccourcir les chemins lors des déplacement et en outre d'obtenir un meilleur niveau de paramétrage dans les scripts. La localisation exacte du répertoire d'un utilisateur est indiquée dans l'un des fichiers de configuration du système.
Ainsi, la commande précédente aurait pu s'écrire de différentes façons :

paul@linux:/tmp/>***cd ~***
paul@linux:/home/paul/>

paul@linux:/tmp/>***cd ~paul***
paul@linux:/home/paul/>

Ou encore plus simplement, car sans paramètre ***cd*** est équivalent à ***cd ~*** :
paul@linux:/tmp/>***cd***
paul@linux:/home/paul/>

De façon générale, dans un tel cas, l'affichage du prompt sera légèrement différent pour en optimiser la longueur, vous aurez plutôt :
paul@linux:/tmp/>***cd***
paul@linux:~>

La commande ***cd*** peut être utilisée avec comme argument « - », ce qui signifie alors « revenir au répertoire précédent » ; ce qui donne par exemple :

paul@linux:/tmp/>***cd***

paul@linux:~>***cd -***
paul@linux:/tmp/>

Le déplacement relatif se fera en utilisant la même commande mais en indiquant le chemin par rapport au répertoire courant plutôt que par rapport à la racine. Le répertoire « **..** » sera alors utilisé pour indiquer qu'il faut remonter d'un niveau. Reprenons l'exemple précédent avec un chemin relatif:

paul@linux:/tmp/>***cd ../home/paul***
paul@linux:~>

Avec « **..** » nous allons remonter d'un niveau pour nous situer dans « / » puis nous allons descendre dans « **home** » et enfin dans « **paul** ». Il est possible de préciser que l'on part du répertoire courant à l'aide du « **.** » mais ce comportement est celui par défaut ; il sera plutôt utilisé, nous le verrons par la suite, pour lancer des programmes ou copier des fichiers.

Nous n'avons pas encore abordé le lien existant entre les partitions d'un disque dur ou même comment naviguer entre plusieurs disques durs. La raison en est simple, cette notion n'existe simplement pas, pas qu'il soit impossible d'avoir plusieurs disques, bien au contraire, mais du fait qu'une partition ou un disque peut être attachée en tout point de l'arborescence. Ainsi derrière chaque répertoire peut se cacher une source physique différente, qui peut être une partition d'un disque comme un système réseau partagé. Ce principe est appelé montage ; il est associé à la commande ***mount*** qui, lancée sans paramètres, vous affichera la liste des montages en cours. Nous n'irons pas dans les détails de l'utilisation de cette commande ici qui pour l'instant s'avère légèrement compliquée.

Plus de détails sur ces fameux fichiers

Maintenant que nous avons vu comment nous déplacer dans l'arborescence, nous allons observer de plus près les fichiers dont nous avons tant parlés. La commande associée à l'affichage du contenu d'un répertoire est ***ls*** ; pour ***L***I***S***T.
Par défaut cette commande affiche simplement en colonne la liste des fichiers du répertoire courant:

paul@linux:~/iut$ ls
system tsignal wifi

Il est possible de lui adjoindre de nombreuses options, consultables au travers du *man,* mais dont les plus usuelles sont les suivantes :

Pour commencer, l'option **-l** est particulièrement intéressante à étudier : elle permet un affichage en détail des données associées à chaque fichiers :

paul@linux:~/iut$ ls -l
drwxr-xr-x 4 paul paul 136 Mar 22 21:28 system
drwxr-xr-x 7 paul paul 238 Mar 17 08:13 tsignal
-rw-r--r-- 1 paul paul 41 Mar 2 19:53 wifi

Le premier caractère nous informe sur le type du fichier, ainsi, la présence d'un **d** en début de ligne nous indique que le fichier en question est en réalité un répertoire (**d**irectory), ce premier caractère sera un « **-** » dans le cas d'un fichier ordinaire. D'autres types de fichiers existent pour indiquer un périphérique de type blocs « **b** » ou caractères « **c** »... Le symbole « **l** » indiquera un lien symbolique, cas un peu particulier que nous verrons un peu plus loin.

Le premier groupe d'informations nous indique ainsi quelles sont les autorisations associées à chaque fichier ; ces informations ne sont compréhensibles qu'à partir de la 3e et 4e zone, ici « paul paul » indiquant respectivement l'utilisateur puis le groupe propriétaire du fichier.

La notion de droit est un élément important du système d'exploitation, qui, comme nous l'avons vu, doit partager les ressources entre plusieurs utilisateurs, il n'est alors pas possible de gérer des fichiers sans en gérer les droits au risque de bafouer toutes règles de confidentialité. C'est ainsi que chaque fichier à un propriétaire. Le travail en équipe se faisant, les fichiers doivent souvent être partagés entre plusieurs personnes si bien qu'est apparu en même temps la notion de groupe.
Un groupe est un ensemble d'utilisateurs et chaque utilisateur est dans au moins un groupe. Ce premier groupe est appelé groupe principal de l'utilisateur, il s'agira de son groupe par défaut. Les autres groupes sont dit groupes secondaires.

Ainsi, chaque utilisateur rangé dans son groupe aura sur les fichiers des autorisations comme des restrictions. Celles-ci sont décrites dans la liste de lettres constituant le premier bloc (à l'exception du premier caractère comme nous l'avons vu). Ces lettres se lisent par groupe de trois et à chacune correspond un droit spécifique. Le premier caractère du groupe de 3, qui peut être « **r** » ou « **-** » indique que le fichier peut être lu (**r**ead) par le propriétaire du fichier lui-même. Le second peut être « **w** » ou « **-** » indiquant que le propriétaire a le droit de l'écrire (le modifier) et le dernier « **x** » ou « **-** » indiquant que le fichier peut être exécuté par son propriétaire. Dans ces différentes combinaisons, il faudra comprendre que la présence du « **-** » indique que le droit en question sera retiré. Ainsi, pour être plus clair, les séries suivantes signifient :

rwx------	- le propriétaire peut lire/écrire/exécuter le fichier
r-x-------	- le propriétaire peut lire/exécuter le fichier, mais son écriture est impossible
r--------	- le propriétaire ne pourra que lire le fichier

Les deux blocs suivants fonctionnent de manière identique, toutefois, ils ne concernent pas les autorisations liées au propriétaire mais celles liées au groupe auquel appartient le fichier pour le second bloc et aux autres pour le dernier. Il faut entendre par « autres » tout utilisateur n'appartenant pas au groupe propriétaire du fichier et autre que le propriétaire lui-même. Pour illustrer cela voici quelques autres exemples :

r--r--r--	- tout le monde peut lire et seulement lire le fichier
------r--	- tout le monde peut lire le fichier sauf l'utilisateur propriétaire et tous les utilisateurs dont le groupe est le même que le groupe propriétaire.
rw-r-----	- le propriétaire peut lire et écrire ce fichier, les autres membres de son groupe peuvent eux, simplement le lire, les autres ne pourront rien en faire.

Nous reviendrons par la suite là-dessus dans un chapitre consacré, plus en détails, aux utilisateurs et à la gestion des droits, continuons donc l'analyse de la fonction ***ls*** avec pour option **-l**. Le premier des nombres, ci-dessous en gras indique le nombre de références à cet élément.

drwxr-xr-x **4** paul paul 136 Mar 22 21:28 system

Ainsi, il nous indique dans le cas d'un répertoire le nombre d'entrées existantes dans ce répertoire.

Le second nombre, ci-dessus souligné, indique la taille occupée par ce fichier, en octets. Suivent ensuite la date et l'heure de dernière modification du fichier. La ligne se terminera par le nom du fichier lui même.

La seconde des options intéressante est l'option **-a** qui permet d'afficher tous les fichiers (**a**ll), y compris, donc, les fichiers dit cachés. Un fichier est caché dès lors que son nom commence

par le caractère « **.** ». Cette façon de faire n'est en rien une solution de sécurité puisque la parade du « ls -a » est connue de tous mais une solution simple pour éclaircir des listes de fichiers un peu longue. C'est ainsi que si vous listez le contenu intégral de votre répertoire personnel, vous trouverez de nombreux fichiers de configuration préfixés d'un « **.** », sans que ceux-ci encombrent un affichage plus classique par ***ls.***

Outre ces options, la commande ***ls*** attend en argument facultatif un ou plusieurs noms de fichiers (ou répertoires) dans ce cas, seront affichées les mêmes informations que celles vues précédemment mais pour ces fichiers/répertoires. Le nom du fichier pourra être indiqué de façon relative ou absolue comme nous l'avons vu précédemment.

D'autres options peuvent être très utiles à connaître, telles que **-d** qui permet d'afficher les informations d'un répertoire sans pour autant lister son contenu, ce qui est le comportement par défaut de la commande. Enfin, l'option **-R** permettra un affichage récursif, ainsi le contenu des répertoires ainsi que de tous les sous-répertoires seront listés à l'écran.

Bien que très appréciée des étudiants, cette dernière option ne m'aura jamais bien été d'un grand secours car le formatage de sa sortie n'est pas très pratique, le recours à la commande ***find*** que nous verrons par la suite a toujours mieux porté ses fruits.

Nous avons évoqué précédemment la notion de liens symboliques lorsque nous avons abordé les types de fichiers. Les liens, de façon générale, consistent à faire pointer plusieurs noms de fichiers (noms qui peuvent être différents, situés dans des répertoires différents) vers un même ensemble de données. Il existe deux types de liens : le lien dit « en dur » ou lien classique et le lien symbolique. Pour comprendre les liens, il faut comprendre le stockage des informations sur le disque dur. L'espace disque se décompose en deux grandes parties : la table d'allocation et l'espace de stockage. Le principe est le même que celui d'une bibliothèque : il y a d'un côté les livres (= les données) et d'un autre côté un index (catalogue) qui permet de retrouver le livre que l'on cherche le plus directement possible. Un fichier est une entrée dans le catalogue et cette entrée pointe sur un ensemble d'espaces de stockage de la seconde partie.
Le principe du lien est de permettre la création de plusieurs entrées dans la table d'allocation (aussi appelée FAT (File Allocation Table)) pouvant pointer, toutes, le même espace de stockage. Cela revient à donner plusieurs noms à un même ouvrage. Le lien à plusieurs avantages : il permet d'éviter des copies et donc gagner de la place, il permet aussi de s'assurer que si plusieurs entrées existent dans le catalogue, la modification de l'une d'entre-elles sera bien reportée dans toutes. Le lien classique fonctionnant au niveau du système de fichier (File System = FS) il n'est pas possible de créer de liens vers des données issues d'un système de fichier différent.
Cette restriction conduit souvent à l'utilisation privilégiée des liens symbolique. Le lien symbolique est une simple redirection, c'est un peu la même chose que le raccourci de Windows. Le principe est d'indiquer au système que le fichier *A* du répertoire *R1* est en fait situé dans le répertoire *R2* sous le nom *B*... Concrètement, la commande **ls** représente les liens comme suit :

```
ls -l
lrwxr-xr-x 1  paul paul      4    lienSymbolique -> fichierLié
-rw-r- -r--  2  paul paul     10    lienClassique
-rw-r- -r--  2  paul paul     10    fichierLié
```

Tout d'abord le lien symbolique : il s'identifie par le type du fichier indiqué par le premier des caractères : ici « **l** » le *bash* représente ce lien en indiquant que le fichier s'appelle « lienSymbolique » mais qu'il s'agit en fait d'une redirection vers « fichierLié ».

Pour ce qui est du lien classique, on ne remarquera pas de différence entre les deux fichiers, c'est normal ... dans un lien classique, il n'y a plus de notion de hiérarchie des liens, ou de source, ce sont

deux entrées de la FAT qui sont équivalentes et pointent vers les mêmes données... Ce qui nous indique la présence d'un lien, c'est le chiffre en seconde position. Le « **2** » indique le nombre de références sur les données.

Le comportement des liens symboliques et des liens classiques va être très différent lors de la suppression d'un fichier : si le fichier « *fichierLié »* est supprimé, le lien classique sera préservé et il sera possible de modifier/lire les données sans soucis. En réalité, lors de l'effacement, c'est l'entrée dans la FAT qui va être supprimée et non les données. Ce n'est que lorsqu'il n'y aura plus d'entrées dans la FAT qu'il ne sera plus possible d'accéder aux données et que la zone de stockage pourra être libérée pour d'autres fichiers.
La suppression de « *fichierLié »* a un impact tout autre sur le « *lienSymbolique* », le fait que la cible n'existe plus rompt le lien, ce changement n'aura pas d'effet apparent sur le lien (selon le shell utilisé) mais rendra l'accès aux données impossible.

La création d'un lien se fait à l'aide de la commande « **ln** » dont la syntaxe est :

ln [-s] source destination

ln fichierLié lienClassique => création d'un lien classique
ln -s fichierLié lienSymbolique => création d'un lien symbolique

Les liens symboliques peuvent pointer n'importe quel fichier situé n'importe où dans l'arborescence, sans se soucier de l'organisation des partitions, file system...

Quelques manipulations avec le *bash*

Le **bash**, comme nous l'avons vu, est un shell, c'est en général celui par défaut sous Linux. Chacun aura son avis, mais pour moi c'est le plus convivial et ergonomique, c'est sans doute pourquoi il est retenu comme shell par défaut.

Il est important de bien savoir manier ce shell, simplement pour gagner du temps et ne pas rendre l'usage de la console rébarbatif... Ainsi quelques touches ont un usage qui est à maîtriser. Commençons par les flèches haut et bas, elles permettent de se déplacer dans l'historique des commandes, il sera ainsi possible de rappeler les dernières lignes validées, de les modifier et les ré-exécuter.
Vient ensuite la touche de tabulation, cette touche permet d'utiliser la complétion automatique. Ainsi, la suite de lettres « **ech** » suivie d'un appui sur **tab** devrait conduire à l'inscription automatique de la commande ***echo***; cette complétion sera très utile pour parcourir l'arborescence des fichiers. Ainsi, « **/ho » tab « /pa » tab** me permettra avec un moindre effort de me rendre dans mon répertoire personnel : **/ho**me**/pa**ul

Ces quelques touches sont les principales à connaître et si ces fonctions existent sur la plupart des shells modernes, elles ne sont pas toujours utilisables de la même façon, ainsi sur les serveurs IBM sous ksh (en mode vi), par exemple, l'historique est parcouru par ESC puis « - » et « + » et la complétion (qui ne marche que lorsqu'une seule réponse est possible) se fait par ALTGR + \ .. pas simple !

Outre ces astuces spécifiques aux shells, il existe quelques commandes compatibles utiles à connaître. Entre autres, la commande ***history*** qui permet d'afficher l'historique de toutes les commandes tapées, précédées d'un numéro d'ordre. Ce numéro permettra de rappeler chacune d'elles en le précédant du caractère « ***!*** ». Ainsi, si history donne le résultat suivant :

paul@linux:/tmp>***history***
1 echo HelloWorld!!!

2 echo "Hello World"

Il sera possible de redemander l'exécution de la seconde commande en tapant :

paul@linux:/tmp>***!2***
Hello World

Les shells manipulent des variables, celles-ci peuvent stocker toutes chaînes de caractères. Un script pourra ensuite les interpréter comme des nombres si nécessaire. Il n'y a donc pas de typage des variable comme en C, Java ou autres langages évolués, ni de déclaration préalable à faire. Une variable commence toujours par une lettre et se compose de caractères alpha-numériques.
L'affectation d'une variable se fait simplement à l'aide de l'opérateur « = » :

maVariable='maValeur'

Vous ferez attention à ce qu'il n'y ait aucun espace ni avant, ni après le symbole d'affectation.

Les variables sont sensibles à la case, ainsi *maVariable* et *MAVARIABLE* sont deux variables bien distinctes.

Une variable est par défaut locale, c'est à dire qu'elle n'est visible que pour le shell dans lequel elle a été déclarée et qu'elle ne sera pas transmise aux programmes lancés depuis ce shell. Pour qu'elle soit transmise, il est nécessaire de « l'exporter » (on pourrait dire publier) à l'aide de la commande « export ».

export maVariable='maValeur'

Ainsi, la variable maVariable pourra être lue par n'importe quel programme lancé depuis le shell. Je précise bien « lu » car les sous-programmes ne pourront pas la modifier.
Il est possible de consulter la liste des variables exportées en utilisant la commande « **env** ». Cette commande signifie « environnement », en effet, on dit que les variables sont stockées dans l'environnement du shell.

Une variable sera supprimée par la commande **unset.**

unset maVariable

Il faudra différencier les notions de variable vide (maVariable='') et d'absence de variable (unset maVariable).

L'utilisation d'une variable **se fait toujours en faisant précéder le nom de la variable par le symbole $**. Ainsi pour afficher la valeur de *maVariable*, nous écrirons :

echo $maVariable

Rq : lors d'une affectation, vous ne devez jamais trouver le symbole $

Une variable peut aussi contenir des nombres, il faudra toutefois les traiter de façon particulière pour qu'ils soient considérés comme tels, ainsi :

a=1
b=a+1
echo $b

Donnera le résultat suivant : « **1+1** » et non 2 => les chiffres sont considérés comme des caractères et non des nombres.

Pour que le calcul se fasse, il faudra écrire les lignes ainsi :
a=1
b=$(($a+1))

Dans ce cas, c'est un opérateur particulier $(( )) qui est appelé, cette commande étant prévue pour réaliser des calculs mathématiques sur des nombres entiers. Le résultat sera donc « **2** ».

Il existe de nombreuses variables définies au lancement du système et des shells. Les plus courantes sont :

PS1	=>	définit la forme du prompt
PATH	=>	définit les chemins par défaut dans lesquels trouver les programmes

Rq : Lors de l'utilisation de variables, il pourra être nécessaire de commencer et terminer leur nom par les caractères « { } ». En effet, si nous souhaitons afficher un texte comme "il est 8h" où 8 sera issu d'une variable « *heure* » sans mettre d'espace entre le chiffre et la lettre « h », alors nous devrons écrire la ligne ainsi : **echo ${**heure}h. Le shell reconnaitra alors la variable *heure* grâce aux délimiteurs. Sans cela, nous aurions écrit : **echo** $heureh ; le shell aurait alors reconnu une variable « *heureh* » qu'il n'aurait pu évaluer correctement.

L'éditeur de texte VI, où comment utilement traumatiser les jeunes informaticiens

Je fais partie de ces quelques convaincus qui se disent qu'en informatique une tête bien faite vaut mieux que mille têtes bien pleines, tellement la programmation est une remise en cause perpétuelle, l'énoncé à résoudre étant chaque fois si différent qu'il n'est pas de méthode pour écrire les programmes à notre place... Je veux dire par là que celui qui penserait que réussir en informatique consiste à apprendre une quantité faramineuse de commandes et d'options ne ferait que perdre son temps ; incapable de s'en sortir face à une situation nouvelle. Réussir en informatique c'est à mon sens percer ses mécanismes, comprendre toujours le pourquoi du comment et posséder suffisamment de repères pour trouver dans toutes situations un point de départ duquel par raisonnement, il sera possible de progresser petit à petit vers la solution escomptée...
Tout ça pour vous dire plus simplement, que malgré la longue liste de commandes et options que nous allons voir, il n'est pas nécessaire de toutes les mémoriser. Le point réellement important est de savoir qu'elles existent car c'est par l'interaction de multiples petites fonctions que vous résoudrez vos futurs problèmes informatiques ; la clef étant d'être capable de les retrouver, plus généralement être capable de trouver un solution toute faite en utilisant les bons moyens et les bons mots clefs. Tout ceci étant dit, il est aussi nécessaire de connaître un minimum de choses par coeur, non seulement pour avoir quelques points de départ à toute analyse, mais aussi pour gagner du temps. Ainsi, si ***man*** vous donne accès à toute la documentation, encore faut-il se rappeler de cette commande. Par là je veux vous amener à comprendre le pourquoi du comment nous allons dans un instant replonger dans ce qui vous semblera être la préhistoire de l'informatique mais qui reste pourtant mon quotidien d'informaticien avant de devenir bientôt le vôtre : ***vi.***

Il fut un temps où ni la souris, ni les menus et encore moins les environnements graphiques existaient. Les informaticiens ont donc dû remplacer tout ce que nous utilisons communément par l'usage intensif du clavier et ce pour une application des plus banales : un éditeur de texte. Voilà donc ce qu'est ***vi*** : un éditeur de texte où toutes les commandes sont accessibles au clavier.
Pourquoi tant de haine envers les nouvelles générations d'informaticiens, sont elles obligées de subir les mêmes supplices que leurs pères ? Simplement non... et simplement, si l'enseignement de ***vi*** reste un point important c'est parce qu'aujourd'hui encore on ne fait pas beaucoup mieux en ligne de commande. ***Vi*** est en effet le seul éditeur que vous trouverez quel que soit le système, c'est le seul éditeur dont le fonctionnement n'est pas dépendant du clavier sur lequel il est utilisé. Enfin, vous l'auriez sans doute découvert un jour : le modernisme de Linux et de son bash sont loin d'avoir été intégrés à tous les UNIX, ainsi le rappel de commande pour le déplacement du curseur sur une ligne de commande se fait encore souvent en *mode vi*. Il est donc d'une grande utilité d'en connaître le fonctionnement. Enfin, si à l'époque du mode graphique ce type d'éditeur est encore très utilisé c'est aussi parce que l'administration de serveur se fait **toujours** à distance et que sur un réseau, aussi rapide qu'il soit, le mode texte n'a pas d'égal.

Les connaisseurs remarqueront que je n'ai pas évoqué l'autre éditeur qu'est *emacs* et sa magnifique version graphique *xemacs*... Il y a sous UNIX les adorateurs de la touche ESC (vous verrez pourquoi) et ceux du control+k dont je ne fais bien sûr pas partie ; par ailleurs si ***vi*** reste omniprésent, il n'en est rien *d'emacs* et encore moins de *xemacs* où utiliser une solution graphique aussi pauvre ne présente plus aujourd'hui aucun intérêt à mon sens ; si ce n'est d'être léger.

Bref, entrons dans le vif du sujet : l'éditeur ***vi.*** Son lancement est plutôt simple, il suffit de taper la commande ***vi*** voire de la faire suivre d'un nom de fichier ; auquel cas le contenu de ce dernier s'affichera dans la console. Jusque là, tout va bien... essayez alors de taper du texte et vous pouvez switcher en mode panique !!! comment quitter ce truc !

Bon ! démystifions tout ça ! **Vi** ne possédant pas de menu, il possède un mode *menu* que l'on appelle plus communément le **mode commande.** Ce mode est celui par défaut. C'est un peu comme si dans une application graphique vous étiez dans la barre de menu après un click sur l'une de ses entrées ; sauf que là rien n'est bien visible. En mode commande, vous disposez d'un certain nombre de touches vous permettant d'avoir accès à toutes les options classiques d'un éditeur. Ainsi en mode commande il suffit de taper sur une des touches suivantes pour obtenir le résultat associé :

:w	-	enregistrer
:w *nom*	-	enregistrer sous
:q	-	quitter
:q!	-	quitter sans sauver
:r *nom*	-	ouvrir un fichier

Il est ainsi aussi possible d'agir sur le texte présent dans l'éditeur :

x	-	supprimer le caractère courant
u	-	abandonner les dernières modifications (undo)
dd	-	supprimer la ligne courante
*num***dd**	-	supprimer *num* lignes à partir de la ligne courante
*num***yy**	-	copier *num* lignes à partir de la ligne courante
P	-	coller
J	-	supprimer le retour charriot (remonte la ligne suivante sur la courante)

Comme il est possible de se déplacer dans le texte :

$	-	aller en fin de ligne
^	-	aller en début de ligne
l	-	se déplacer à droite (utile quand les flèches ne sont pas reconnues)
h	-	se déplacer à gauche
j	-	aller à la ligne suivante
k	-	aller à la ligne précédente

Enfin, il y a plusieurs façon de quitter le mode commande pour passer en mode édition et ainsi modifier le texte :

i	-	permet de passer en mode insertion à la position courante
a	-	permet de passer en mode insertion après la position courante
A	-	permet de passer en mode insertion à partir de la fin de ligne
R	-	permet de passer en mode remplacement

Le retour en mode commande se fait par l'usage de la touche **ESC**. Sur les versions récentes de ***vi*** le mode édition est indiqué par le mot « INSERT » ou « REPLACE » en bas de fenêtre.

Outre ces quelques commandes simples, il est possible de faire un usage très évolué de ***vi***. Voici quelques autres commandes utilisables dans le mode ad-hoc :

:set ai	-	active l'indentation automatique

:set noai	-	désactive cette même indentation
:set nu	-	active l'affichage du numéro de la ligne
:set nonu	-	désactive ce même affichage
:*num*	-	aller à la ligne *num*
/*mot*	-	permet de recherche un mot dans le texte
/	-	permet de passer à l'occurrence suivante du même mot

%s /*rech*/*remp*/**g** recherche et remplace toutes les occurrences d'une chaîne par une autre.

Voici donc quelques commandes de cet outil, le manuel vous en donnera de multiples autres. Toutefois, celles-ci sont les plus couramment utilisées, ce qui sous-entend qu'il n'y a pas d'autre choix que de les apprendre !!!

Si, ce sera votre cas, vous avez vraiment l'impression d'avoir des difficultés avec vi, sachez que c'est normal et que pour ma part j'ai longtemps lutté contre avant d'être convaincu puis prosélyte à son égard ! Je vous le redis, ***vi*** est une sorte de passage obligé dans la vie d'un informaticien travaillant sur système UNIX, c'est aussi un éditeur très efficace dès lors que l'on en maîtrise l'utilisation. Je veux dire par là qu'avec un peu d'expérience la plupart des opérations au clavier sont beaucoup plus rapides qu'avec l'usage d'un éditeur graphique à la souris et puis ... CRTL+S ou ESC :w ... est-ce si différent ?!? Vous voilà en tous cas armés pour la suite de nos manipulations.

Gestion des droits utilisateurs, ou encore de la sécurité

UNIX est un système initialement conçu pour être multi-utilisateurs, ce fait signifie que plusieurs personnes peuvent en même temps utiliser l'ordinateur, lancer des programmes différents, chacun de leur côté, sans que ceux-ci n'impactent les autres... Bien que votre usage de Linux, dans les premières semaines, ne vous amènera sans doute pas en prendre réellement conscience, un système UNIX n'est pas initialement conçu pour se limiter à une personne face à son micro-ordinateur, mais plutôt à un groupe d'utilisateurs plus ou moins lointains, interagissant au travers d'un terminal...

Le concept de terminal vous semble peut être trouble. Imaginez un ordinateur apte à peu de choses de plus que l'affichage d'une image et la transmission des mouvements de la souris et des pressions des touches de clavier : c'est un terminal. Quelqu'uns auront vu un minitel ou en auront, au moins, entendu parler. Le minitel était l'exemple typique du terminal avec son origine direct : proposer une solution à très bas coût permettant de se connecter à un serveur, lui, très cher. De nos jours cette notion a-t-elle encore du sens ? Il n'en fait aucun doute car le terminal offre de très nombreux avantages en terme de maintenance et toujours de coûts. Le terminal d'aujourd'hui s'oriente toutefois vers une forme plus intelligente : capable d'exécuter un navigateur web et une JVM pour permettre l'exécution des applications d'entreprise de type client léger ou riche (web).

L'objet de ce chapitre étant de comprendre comment UNIX gère ses utilisateurs, je vous propose donc de revenir à nos utilisateurs.

Gérer plusieurs utilisateurs demande tout d'abord de les identifier. Cette identification se fait à l'ouverture de session par une procédure de « login ». Un utilisateur possède un pseudonyme ou « login » accompagné d'un mot de passe. A la connexion, le système vérifiera si ce couple login, mot de passe correspond bien à un couple attendu. Pour cela, le système se repose sur un à deux fichiers de configuration.

Le premier de ces fichiers est **/etc/passwd** : il s'agit d'une sorte de base de données des utilisateurs sous la forme d'un fichier plat (une ligne correspond à un enregistrement). Chaque ligne suit le format suivant :

login : mot de passe : uid : gid : gecos : home : shell de connexion

Le **login** est un pseudonyme utilisé pour identifier de façon unique un utilisateur.

Le **mot de passe** est un enregistrement crypté du mot de passe de l'utilisateur.

L'**UID** est un nombre unique identifiant cet utilisateur. En effet, un système informatique serait bien ennuyé à utiliser partout la chaîne de caractères du login pour identifier les éléments, ainsi a chaque login on associera un **uid** qui sera utilisé partout. Le login n'est lui utilisé que pour la connexion et l'affichage.

Le **GID** est un identifiant de groupe, nous verrons cette notion par la suite.

Le champ **GECOS** est un champ plus ou moins libre dans lequel on entrera le nom complet de la personne, son numéro de téléphone et ce genre d'information permettant de l'identifier physiquement.

Le champ **home** indique le répertoire de l'utilisateur, c'est dans ce fichier qu'il sera définit et ici seulement.

Enfin le champ **shell** indique quelle commande est à exécuter lors du login. Il s'agit en général d'un shell (comme le bash), mais il peut s'agir du programme **false** par exemple pour empêcher l'ouverture d'un shell sur un compte qui ne correspondrait pas à un humain mais plutôt à une application telle qu'un robot. Il peut aussi envisager de demander l'exécution de n'importe quel programme à la connexion, en remplacement du shell standard.

Passons quelques temps sur la notion de mot de passe, critique en sécurité. Il est à noter que pour des raisons de sécurité, il est impossible de retrouver le mot de passe d'un utilisateur, en fait, le système lui même l'ignore. Le principe repose donc sur son remplacement par une chaîne calculée à partir du mot de passe. Lors de la connexion, le mot de passe sera donc vérifié par comparaison des deux mots de passe cryptés et non par comparaison des mots de passe eux-mêmes. L'intérêt de cette solution réside dans le fait qu'il est presque impossible de retrouver le mot de passe initial à partir de la chaîne cryptée, ainsi, même si le mot de passe crypté est connu, il n'est pas possible de l'utiliser pour se connecter.

Pas possible, enfin pas tout à fait, le crack de mot de passe est malgré tout une activité répandue, mais celle-ci ne repose pas sur l'attaque directe de la chaîne cryptée, mais plutôt sur l'attaque du mot de passe initial. L'idée étant d'essayer un nombre limité de combinaisons issues d'un dictionnaire adjoint à quelques algorithme pour générer des mots de passe (en clair) potentiels, crypter ceux-ci puis comparer le résultat à la chaîne trouvée sur le système. Si elles sont identiques, la clef est cassée !

Le principe de sécurité repose sur le fait que le temps nécessaire à calculer une chaîne cryptée est suffisamment long pour que le calcul de millions de combinaisons rendent le procédé caduque par l'expiration ou l'obsolescence du mot de passe. Reste que pour que la sécurité soit assurée, il faut utiliser un mot de passe que l'on ne trouvera pas dans un dictionnaire. Là dessus chacun aura sa méthode et proposer une méthode la rendrait par définition caduque.

Les méthodes courantes consistent à ne simplement pas utiliser de mots mais une suite de signes sans sens particulier. Cette méthode à la désagréable conséquence de générer autour des écrans l'apparition de *Post-it* recouverts de mots de passe... Bref sans doute sûre, mais très dangereuse.

La clef d'un mot de passe est donc la capacité de l'utilisateur à le retenir, alliée à une construction complexe. Le mieux sera alors de se baser sur un mot de vocabulaire courant que l'on va écrire de sorte à le rendre inhumain. Remplaçons par exemple « lapin » par « lap1 », Ajoutons quelques signes « l*a*p*1 » est nous avons un mot de passe simple à retenir mais que l'on ne trouvera pas dans un dictionnaire. Reste que ce mot de passe est beaucoup trop court pour être sûr. En effet lorsqu'une attaque par dictionnaire n'aboutit pas, il reste la méthode « brute force » qui consiste à tester toutes les combinaisons possibles et là, la longueur du mot de passe primera.

Bien que je puisse remplir des pages sur ce sujet, je vais tâcher de me recentrer sur UNIX. Comme nous venons de l'illustrer, la sécurité de mots de passe sera mise à mal dès lors que sa version cryptée sera connue. Or, initialement un système UNIX se doit de permettre à tout utilisateur de

consulter le ficher « /ect/passwd » de sorte à ce que chacun puisse se connecter, chercher le home d'un autre utilisateur (cd ~autreUtilisateur)... Ainsi tout possesseur d'un compte sur le système est susceptible de pouvoir extraire la liste des mots de passe cryptés. Au début sans risque devant le temps nécessaire au cryptage d'un mot de passe, aujourd'hui lourd de conséquences avec les capacités de calculs que nous possédons. Pour cette raison est apparue une mécanique dite des « **shadow password** » dont le but est de restreindre l'accès aux mots de passe, sans pour autant limiter l'accès au fichier historique « /etc/passwd ». Il s'agit de retirer du fichier habituel (passwd) les mots de passe pour les stocker dans un second fichier de même format qui s'appellera « /etc/shadow » , « /etc/passwd~ » ou autre, selon les systèmes. Le premier fichier sera publique alors que le second ne pourra être ouvert que par le système lors des opérations de login.

Ce système est important pour votre culture générale mais c'est aussi un point à ne pas négliger lorsque l'on souhaitera manuellement éditer le fichier « /etc/passwd » : il faudra veiller à mettre en cohérence son image privé (shadow). Plus généralement, lorsque l'on souhaitera modifier le fichier passwd, il faudra désactiver le système de shadow par la commande « pwunconv » puis réactiver ceux-ci après modification par la commande « pwconv ». Seul root peut opérer ce genre de commandes, cela va de soit.

Voilà pour ce qui touche à la définition des utilisateurs. Vous aurez remarqué que nous n'avons nullement parlé de ce qu'un utilisateur peut ou non faire. Ce n'est donc pas dans ce fichier que nous trouverons la réponse à cette question, mais plutôt, partiellement, dans un second : « /etc/group ».
Le fichier de groupes permet de définir, comme son nom le laisse facilement penser, des groupes d'utilisateurs.
Un utilisateur UNIX sera obligatoirement dans un groupe principal, ce groupe est défini dans le fichier « /etc/passwd » sous le champ GID que nous avons vu précédemment. Le GID étant l'identifiant numérique du groupe, au même titre que l'UID correspond à l'utilisateur. Le groupe principal sera celui choisi par défaut lors de la création de fichiers, comme nous le verrons par la suite.
Le fichier « /etc/group » permet de déclarer les groupes en associant à des chaînes usuelles (telle que « user » par exemple) le GID utilisé par le système. Il permet en outre d'ajouter à chaque groupe des utilisateurs supplémentaires. Ces associations seront alors autant de groupes secondaires auxquels l'utilisateur appartiendra.

La notion de hiérarchie de groupes n'existe pas sur UNIX, ainsi on devra généralement créer autant de groupes que de fonctions/programmes que l'on souhaitera contrôler et l'on associera chaque utilisateur à chacun des groupes dont il aura besoin. Cette gestion historique ne rend pas l'administration simple et sur ce point les systèmes Windows sont plus avancés, reste que ce système est complété par celui des ACL sous UNIX et qu'il offre l'avantage d'être clair. Je veux dire que ajouter/retirer des droits se fait sans trop de risques d'oublis alors que les systèmes plus modernes autorisent l'association d'un même droit par de multiples façons, impliquant que le retrait de certains droits pourra être oublié si l'administrateur n'est pas suffisamment rigoureux.

Le fichier « group » intègre lui aussi une notion de mot de passe pour des administrateurs de groupes, mais je n'ai encore jamais vu cette fonctionnalité mise en oeuvre. Ainsi le fichier « group » est lui aussi impacté par la méthode des « shadow password »

La notion de droits et leurs définitions ne se faisant toujours pas dans ce fichier, c'est directement au niveau de chaque programme ou fichier que le droit sera associé sous UNIX. En reprenant le principe d'UNIX selon lequel tout est fichier, ce système permet alors de façon homogène de fixer des restrictions autant sur un fichier que sur un programme que sur un périphérique. Je peux par exemple interdire à des utilisateurs l'utilisation de la carte son de l'ordinateur aussi simplement que la lecture d'un fichier. Pour ce faire chaque fichier du système possède à la fois des droits d'accès et

un propriétaire défini par un user et un groupe.
Ces informations, nous les avons vues précédemment lorsque nous avons étudié la commande « **ls** ». Ainsi nous avions commenté la commande suivante :

```
paul@linux:~/iut$ ls -l
drwxr-xr-x   4 paul  paul  136 Mar 22 21:28 system
drwxr-xr-x   7 paul  paul  238 Mar 17 08:13 tsignal
-rw-r--r--      1 paul  paul   41 Mar  2 19:53   wifi
```

Le premier groupe de signes correspond aux droits sur le fichier, associés au propriétaire, à un groupe donné et aux autres. Les droits étant, pour rappel, composés d'une suite de trois lettres rwx dont la présence indique respectivement, pour un fichier, un droit de lecture (r), d'écriture (w) et d'exécution (x).
Les colonnes 3 et 4 correspondent respectivement à un utilisateur et à un groupe qui sont propriétaires du fichier. Je vous invite à vous référer au chapitre sur « ls -l » pour vous remémorer plus en détail ces points.

Le cas des droits d'un répertoire est particulier et je vous invite à bien le retenir car je vois de très nombreuses personnes sécher de longues minutes devant la suppression d'un fichier sur lequel ils pensent avoir tous les droits, je m'explique :
Les droits d'un répertoire ne signifient pas la même chose que ceux d'un fichier :

- la lecture d'un répertoire correspond à la capacité que l'on a à lister son contenu
- l'écriture d'un répertoire indique la possibilité de créer et de supprimer des fichiers
- l'exécution d'un répertoire étant la possibilité de le traverser, d'y accéder.

Ainsi il est possible d'autoriser l'accès à un répertoire sans en autoriser le listing : **-wx** par exemple.
Revenons en au cas d'école de la suppression d'un fichier. La situation que j'ai souvent pu rencontrer est la suivante : un utilisateur copie un fichier dans un répertoire dont les droits sont les suivants : rwxrwxr-x. Le fichier appartient à son groupe primaire et à lui. Un autre utilisateur, cherchant de la place souhaite alors le supprimer, or il ne fait pas partie du même groupe principal. Les droits qui lui sont donc opposés sont alors ceux de « other » limités à « r-x » signifiant qu'il ne peut ni créer de nouveaux fichiers, ni supprimer ceux des autres. L'utilisateur propriétaire du fichier aura beau mettre un droit « rwxrwxrwx » sur le fichier lui-même, le second utilisateur ne pourra le supprimer.
La seule possibilité qu'aura alors le second utilisateur sera de vider le fichier en l'ouvrant, le vidant et le sauvant puisque cette opération lui est permise au niveau des droits du fichier lui-même. Illustrons cela:

```
drwxrwxr-x paul:paul        repertoire
-rwxrwxrwx paul:paul        fichier
```

L'utilisateur gerard:gerard ne pourra pas supprimer le fichier car il est identifier au niveau du répertoire comme « other » et ses droits sont limités à r-x, lui interdisant création et suppression.
Il pourra par contre lire le fichier et même le modifier puisque les droits du fichier sont rwx pour other.

De façon complémentaire, dans l'exemple suivant le fichier le pourra pas être lu, mais il pourra être supprimé par tout le monde, voire remplacé par un autre !

```
drwxrwxrwx paul:paul      répertoire
-rwx-------               fichier
```

Attention donc à la gestion des droits des répertoires !

Nous voilà au point sur le fonctionnement des droits sur les fichiers, reste à voir comment l'on peut utiliser cela pour gérer notre système et limiter les actions de nos utilisateurs sur le système. Nous allons créer des groupes pour chaque action que nous souhaitons mettre sous contrôle. Prenons

l'exemple de la carte son. Je veux limiter son utilisation a certains utilisateurs seulement :

- Je vais créer un groupe « audio » par exemple.
- Je vais ensuite restreindre les droits sur le périphérique /dev/dsp (ou autre nom) en lui associant les propriétés et droits suivants :

  brwxrwx--- root:audio dsp

  Ainsi, le groupe audio aura tous les droits sur le périphérique alors que « les autres » ne pourrons pas y accéder.

- Il restera à modifier le fichier « /etc/group » pour ajouter au groupe « audio » la liste des utilisateurs autorisés à accéder au périphérique.

Les commandes associées à ces manipulations sur la gestion des droits

Plusieurs commandes UNIX permettent la manipulation des utilisateurs, groupes et droits, la plupart du temps elles seront limitées à l'usage par l'utilisateur « root », pour d'évidentes raisons de droits et de sécurité :

- **adduser** permet de créer de nouveaux utilisateurs sur le système (complète le fichier /etc/passwd et /etc/shadow). Seul root peut l'exécuter puisque lui-seul peut écrire dans le fichier « /etc/passwd ».
- **passwd** permet de changer son mot de passe. Root pourra, lui, changer de mot de passe de n'importe quel utilisateur.
- **chown** permet de changer l'utilisateur et le groupe propriétaire d'un fichier.
  Son usage est le suivant : ***chown*** *utilisateur**:**groupe nomFichier*
  nomFichier appartiendra alors à l'utilisateur *utilisateur* et au groupe *groupe.*
  De façon générale, un utilisateur ne pourra changer le groupe que parmi ceux auxquels il appartient. Cette commande demande donc souvent d'être root pour être employée. L'option la plus courante est « -R » pour indiquer un usage récursif (application à tous les sous-répertoires).
  Il sera possible de changer seulement le groupe par la syntaxe :
  chown :*group nomFichier*
- **chmod** permet de modifier les droits sur un fichier. On indiquera les droits à positionner de différentes façons. Root peut modifier les droits de n'importe quel fichier alors qu'un utilisateur devra se cantonner à ses propres fichiers. Cette commande vous permettra d'autoriser vos amis à accéder à certains de vos fichiers en autorisant la lecture, la modification ou l'exécution.
  La syntaxe est la suivante : ***chmod*** *droits nomFichier*
  La zone *droits* prendra différentes formes :
  - zone(s){+/-/=}droit(s) où zone sera « u » (user) / « g » (group) / « o » (other) / « a » (all) et les droits « r » read / « w » write / « x » execute. Ce qui donne par exemple:
    chmod ug+rw => ajouter les droits read et write à l'utilisateur et au groupe propriétaire.
    chmod a-w => enlever à tous le droit d'écriture.
    chmod u=rwx,g=r => les droits de l'utilisateur propriétaire seront read, write et execution ; ceux du groupe limités au seul « read »
  - La seconde méthode est une notation octale : on code chacun des droits sur un bit ainsi le droit « read » vaut 4, le droit « write » vaut 2 et le droit « execution » vaut 1. On indiquera la valeur que l'on souhaite fixer aux droits du fichier en positionnant trois digits pour respectivement « user » « group » et « other » avec pour valeur la

somme des droits souhaités.
Ainsi, pour donner les droits suivants : rwxr-x--- la chaine de droit sera créée comme suit :

- pour « user » nous souhaitons rwx soit 4+2+1 = 7
- pour « group » nous souhaitons r-x soit 4+0+1 = 5
- pour « other » nous souhaitons --- soit 0+0+0 = 0

La commande sera alors *chmod 750 nomFichier.*

Recherche de fichiers et utilisation de joker

Comment retrouver une aiguille dans une meule de foin ??? En informatique c'est assez simple, il faut décrire ce qu'est l'aiguille par rapport au foin puis lancer la commande adéquate, le système triera comme un grand chaque brin de paille jusqu'à trouver cet intrus !!! Rien de plus simple non ? Cet outil agricole si pratique est la commande « **find** ». Maintenant comment trouver non pas une aiguille mais toutes sortes d'instruments s'en approchant (comme des aiguilles de couture, des aiguilles à tricoter, des perçoirs...) ? Il faut simplement faire une définition plus large de l'aiguille en utilisant des « **jokers** » et le tour sera joué !

Les jokers permettent de définir des remplacements, le plus connu est « * », il remplace dans un shell n'importe quelle suite de caractères. Ainsi, appliqué à la commande ls, il donne « ls * » qui revient à demander la liste de tous les fichiers. Contrairement à « ls » sans option qui donnera aussi la liste de tous les fichiers, « ls * » est en réalité traduit, avant l'exécution de la commande « ls » par « ls *fichier1 fichier2 fichier3 fichier4 fichier5...* » cela signifie que l'interprétation du joker est faite avant même l'appel de la commande : le programme « ls » ne recevra donc jamais comme paramètre « * » mais il recevra en paramètre la liste de tous les fichiers.

Dans cet exemple, l'emploi de « * » n'est pas très intéressant, regardons alors d'autres associations comme « a*.c » qui signifie : tous les fichiers commençant par un « a » et se terminant par « .c » . Cette chaîne fera correspondre « a.c » comme « abcdef.c ». « * » pourra donc remplacer n'importe quelle chaîne, y compris la chaîne vide.

Un autre joker est « ? » il correspond lui à « n'importe quel caractère ». Il pourra être utilisé pour faire une sélection plus restrictive comme par exemple *« rm pic??.jpg »* qui supprimera « pic01.jpg » sans détruire « pic101.jpg » qui ne répond pas au modèle.

Si nous souhaitons être encore plus restrictif, il est possible de lister des caractères souhaités ou non. L'exemple suivant stipule que les caractères suivant « pic » seront obligatoirement des chiffres : « *rm pic[012456789][012456789].jpg* » Les crochets indiquent la sélection dans une liste de caractères possibles. Il est possible d'indiquer une liste de caractères interdits en utilisant le symbole de négation du shell : « ! » ce qui donnera par exemple *pic[012456789][!0123]* indiquant que le second caractère variable peut être tout sauf 0, 1, 2 ou 3.

Le joker étant le comportement par défaut pour ces caractères, l'indication du « caractère * » en lieu et place du « joker * » se fera en protégeant le caractère pas un « \ ». Du coup la commande « ls -l *.c » listera le fichier dont le nom est « *.c » et non la liste des fichiers se terminant par « .c ».

C'est ainsi que l'on peut faire la mauvaise blague consistant à créer un fichier « * » sur le home d'un utilisateur, qui s'empressera de l'effacer par la commande « rm * » et détruira ainsi tous ses fichiers, alors qu'il aurait fallu taper « rm * ».

La commande **find** permet la recherche des fichiers sur le système. Cette commande reçoit des options très diverses permettant de définir très finement l'objet recherché, à partir de son nom, mais aussi de ses attributs comme son type, ses dates, son chemin... Une fois ce filtre activé, il sera

possible de demander l'exécution d'une action pour chacun des élus. Cette action ira du simple affichage à l'exécution d'une commande laissée libre à l'utilisateur. **Find** est un outil très puissant qu'il faut savoir apprivoiser, il vous fera gagner beaucoup de temps dans de nombreuses opérations.

La syntaxe de **find** est la suivante :

find *path [predicats ...]*

Voyons des exemples simples :

find / -name '*.c' => rechercher depuis la racine '/' tous les fichiers se terminant par .c

find ~ -type 'd' -print => rechercher depuis *home* tous les répertoires et les afficher

find / -empty -print => rechercher depuis la racine tous les fichiers vides et les afficher

L'évaluation des prédicats se fait comme si un opérateur ET les séparait, ainsi, dans le dernier exemple, le prédicat « print » n'est évalué que lorsque le prédicat « empty » est valide. Il est possible de cumuler autant de prédicats que nécessaire, ainsi nous pouvons imaginer l'exemple suivant :

find / -name '*.c' –type 'f' –empty -print

Seront alors affichés (-print) les fichiers (-type 'f') vides (-empty) dont le nom se termine par « .c » (-name "*.c"). L'évaluation de chacun des critères se fera de gauche à droite et l'évaluation s'arrêtera dès lors que l'un des critères ne sera pas rempli. Ainsi la commande retiendra un fichier se terminant par « .c », vérifiera qu'il s'agit d'un fichier puis vérifiera qu'il est vide avant de l'afficher.

L'utilisation d'un prédicat particulier : -o permettra de changer le comportement par défaut de type ET logique par un comportement de type OU. Dans ce mode de fonctionnement, le prédicat suivant sera évalué y compris si le prédicat précédent est faux. Ainsi :

find / -empty -o -name '*.c'

affiche les fichiers qui sont vides ou dont le nom se termine par « .c »

Le prédicat -o se comporte comme s'il y avait des parenthèses ajoutées sur l'ensemble des prédicats le précédant et le suivant : *find / -empty –name '*.c' -o -empty -name '*a.c'* peut se lire *find / (-empty –name '*.c') -o (-empty -name '*a.c').*

Cela signifie, lors de l'évaluation de chacun des fichiers, que si le fichier ne satisfait pas l'ensemble des critères du premier groupe, alors il sera évalué par le second groupe. Dans le fonctionnement OU, si le fichier satisfait le premier groupe, alors il ne sera pas évalué par le second groupe.

Un des prédicats fort utile de *find* est **-exec**, ce prédicat permet pour chacun des fichiers retenus par les prédicats précédents d'exécuter une commande. Je pense que personnellement c'est dans ce but que j'utilise le plus la commande *find* ; mon exemple le plus courant concerne la recherche récursive d'un texte dans un ensemble de fichiers. Nous n'avons pas encore vu *grep* disons donc maintenant que son but est d'afficher les lignes d'un fichier contenant la chaîne de caractères passée en argument. Il est alors possible de coupler *find* et *grep:*

find . -name '.c' -exec grep printf {} ";"*

La commande find va alors parcourir l'arborescence des fichiers depuis le répertoire courant et ne retenir que les fichiers se terminant par '.c'. Pour chacun d'eux, elle va exécuter la commande *grep* en remplaçant le symbole *{}* par le nom du fichier. Le dernier élément ";" indique la fin de la commande. Le résultat affichera toutes les lignes de tous les fichiers '.c' contenant le mot clef *printf,* et ce, de façon récursive.

Le prédicat *exec* peut aussi être utilisé pour réaliser un test particulier, il est, par exemple, possible de modifier l'exemple précédent pour que soit affiché non pas les lignes mais le nom des fichiers

contenant *printf*. Notons qu'il existe une façon plus simple de le faire mais c'est pour l'exemple.

find . -name '.c' -exec grep print {} ";" -exec echo {} ";"*

Ici, les fichiers sont parcourus, ceux se terminant pas '.c' sont retenus, la commande *grep* est exécutée comme précédemment. Si celle-ci trouve une occurrence de *printf,* elle l'affiche et retourne *vrai.* Dans ce cas et seulement dans ce cas, la seconde commande *echo* est appelée avec en paramètre toujours le nom du fichier, qui sera alors affiché à l'écran.

Voici enfin quelques prédicats utiles pour terminer la description de la commande *find*:

- anewer *file* - retenir les fichiers plus récents que *file*
- atime *n* - dernier accès il y a exactement *n* jours ; si *n* est de type -*n* alors seront retenus les fichiers accédés depuis *n* jours
- type *t* - type du fichier (d = répertoire, f = fichier, l = lien)
- iname *motif* - comme -name mais non sensible à la case
- path *motif* - recherche le *motif* dans le chemin et non le nom du fichier
- prune - ce prédicat empêche find de descendre dans les répertoires identifiés par les prédicats précédents

Utilisation des quotes

L'univers de la ligne de commande conduit à beaucoup utiliser les symboles de quote, que ce soit les simples quotes ('), les doubles quotes (") ou les antiquotes (`). Chacune a son usage qu'il faut assimiler pour bien les utiliser.

Tout d'abord les antiquotes, elles sont utilisées pour pré-exécuter une commande et utiliser le résultat de celle-ci dans la commande principale:

echo `date` - Dans cette commande, le shell va exécuter la commande *date* en premier car elle est indiquée entre antiquotes. Le résultat de la commande *date* viendra remplacer la partie entre antiquotes et enfin, la commande principale *echo* sera exécutée avec ce nouveau paramètre.

Ce qui revient à ceci:

echo `date`

1 – exécution de *date* qui donne le résultat suivant : *dim oct 7 15:15:36 CEST 2007*

2 – remplacement dans la commande principale : *echo dim oct 7 15:15:36 CEST 2007*

3 – exécution de la commande principale : *echo dim oct 7 15:15:36 CEST 2007*

4 – résultat : *dim oct 7 15:15:36 CEST 2007*

Voyons les simples et double quotes: elles servent à protéger la chaîne de caractères, en permettant par exemple de dissocier les espaces de la séparation d'arguments, ainsi les commandes suivantes seront interprétées différemment :

cp *fichier source.txt fichier destination.txt*

cp '*fichier source.txt*' '*fichier destination.txt*'

La première commande comprend 4 paramètres (ou 4 arguments) : *fichier, source.txt, fichier,*

destination.txt. La seconde n'a que deux paramètres : *fichier source.txt, fichier destination.txt*

La différence entre l'usage des simples et doubles quotes s'explique par les différents symboles qui seront ou non protégés de l'interprétation du shell. Nous avons vu que certains caractères comme les jokers et les variables ($) qui sont interprétés par le shell avant l'exécution de la commande. L'usage des doubles quotes restreindra l'exécution des jokers, ainsi:

ls * - listes tous les fichiers du répertoire courant

ls "*" - affiche éventuellement le fichier « * »

L'utilisation de simples quotes sera, lui, plus restrictif encore : en plus des jokers ce seront les variables dont l'interprétation sera évitée:

echo $LANG - Affiche le contenu de la variable LANG

echo "$LANG" - Affiche le contenu de la variable LANG

echo '$LANG' - Affiche la chaîne $LANG

Outre les variables et les jokers, ce sont aussi les antiquotes et les doubles quotes qui seront protégées par les simples quotes.

Les entrées et sorties d'un programme shell

Attention, ce chapitre va définitivement nous sortir des environnements graphiques qui vous sont sûrement familiers pour nous conduire dans les sous-bassements des systèmes texte... Nous allons répondre aux questions suivantes :

Comment faire pour automatiser la saisie de données par l'utilisateur, de sorte à ce qu'il n'ait plus besoin d'utiliser son clavier ? Comment enregistrer dans un fichier les données générées par un programme plutôt que de les consulter à l'écran ? Comment faire en sorte que les données générées par un programme soient utilisées comme les données d'entrée d'un autre programme ?

Ceux qui répondront « il suffit de programmer ces comportements » auront tout faux et seront privés de souris pendant trois semaines...

En fait ces questions ont de très simples réponses qui ne nécessitent aucune modification des programmes existants : en effet, ces fonctionnalités sont prises en charge par le système d'exploitation lui-même et non pas par le programme. Je m'explique : lorsqu'un programme lit ou écrit des données, il ne le fait pas sur le clavier ou l'écran, il le fait sur des flux de caractères. Ces flux sont gérés par le système. Le flux d'entrée par défaut est le clavier, le flux de sortie par défaut est l'écran (ou plutôt le terminal courant). Il est tout à fait possible d'aiguiller ces flux de façon différente pour qu'ils utilisent par exemple des fichiers (réponse aux question 1 et 2). Il est aussi possible de connecter un flux de sortie sur un flux d'entrée pour répondre à la question 3.

Pour illustration, sachez que derrière l'écran se cachent en réalité deux flux : le flux de sortie standard (aussi appelé *stdout)* et le flux de sortie d'erreurs (aussi appelé *stderr).* Ce dernier flux permet d'envoyer les messages d'erreurs qui pourront ainsi être redirigés différemment du flux de données standards, pour les masquer ou au contraire les mettre en valeur.

La redirection des flux se fait par l'usage de symboles spécifiques :

- « **>** » *fichier* - indique la redirection de la sortie standard vers un *fichier*
- *«* **2>** » *fichier* - indique la redirection de la sortie d'erreur vers un *fichier*

- *« < » fichier* - indique la redirection de l'entrée standard depuis un *fichier*
- *« | »* - indique la redirection de la sortie d'un programme sur l'entrée d'un autre

Voyons plus en détail comment utiliser ces redirections :

find . -name '.c' -print > resultat.txt*

Recherche les fichiers se terminant par '.c' et les affiche, mais cette fois, cet affichage ne se fera pas à l'écran mais dans le fichier *resultat.txt* qui pourra être lu une fois la recherche terminée.

find / -name '.c' -print 2>/dev/null > resultat.txt*

Par rapport à la commande précédente, la recherche se fait dans « / » ; certains répertoires ne peuvent pas être listés et génèreront des erreurs. Ces erreurs seront redirigées vers un périphérique (/dev/null) qui correspond à une sorte de poubelle. Ces messages seront donc perdus. Nous aurions pu les rediriger vers un fichier erreur.txt.

sort < resultat.txt

La commande *sort* permet de trier, par ordre alphabétique, les lignes entrées au clavier. Dans le présent cas d'utilisation, c'est le contenu du fichier *resultat.txt* qui sera trié. Cette seconde commande permet donc de trier le résultat de la première commande (find)

find / -name '.c' -print 2>/dev/null | sort*

L'utilisation de « | » appelé « pipe » en anglais ou « tube » en français permet de joindre les deux commandes pour n'en former plus qu'une seule. Elle permet en outre de s'abstenir de créer le fichier intermédiaire « résultat.txt »

Le système de tubes est un outil très puissant d'UNIX qui permet de mixer de multiples commandes simples pour créer des opérations complexes de manipulation de fichiers par exemple.

Outre le fait d'éliminer les fichiers intermédiaires, le tube offre un autre avantage : plutôt que de traiter les deux parties « recherche » et « trie » séquentiellement (l'une à la suite de l'autre), le tube traite les deux opérations en même temps. Les commandes *find* et *sort* sont en effet lancées par le système en même temps et s'exécutent donc en parallèle. La commande *sort* traitera donc les données de *find* au fur et à mesure où elles seront générées. Il est possible d'enchaîner plusieurs *tubes* les uns à la suite des autres et ainsi enchaîner autant de commandes que nécessaire.

Dans le cadre d'une utilisation normale, le lecteur doit se terminer après l'écrivain, c'est à dire, dans notre exemple, *sort* ne doit normalement se terminer qu'une fois la recherche (*find*) finie. Dans le cas où cette règle ne serait pas respectée, le système retournera un message d'erreur : « Broken pipe » indiquant la rupture prématurée du flux par le lecteur.

D'autres redirections sont possibles :

- *>> fichier* - la sortie standard est redirigée vers *fichier* mais celui-ci n'est pas préalablement vidé. Ainsi, il se complète par les nouvelles sortie générées par le traitement.

- **&>** *fichier* - la sortie standard et la sortie d'erreur sont toutes deux redirigées vers le même *fichier*
- **>&2** - la sortie d'erreur est redirigée vers la sortie standard

Cette dernière redirection est la réponse à la question que vous n'aurez pas manqué de vous poser : « Comment fait-on pour écrire un message d'erreur ? ». Son utilisation sera par exemple :

echo 'Erreur grave: ....' >&2

Quelques commandes shell très utilisées

Le shell est un outil très puissant et simple d'utilisation pour traiter des fichiers textes, aussi gros qu'ils soient. A titre d'exemple, vérifier que la dernière ligne non vide d'un fichier contient le texte « fin fichier » peut s'écrire en shell sur environ 3 lignes là où en C par exemple il en faudrait plus de trois, rien que pour ouvrir le fichier. Le shell n'est pas un environnent très rapide (bien qu'il fasse appel à des programmes compilés) mais très efficace. Or pour l'utiliser efficacement, il est nécessaire de connaître un minimum de commandes qui pourront être utilisées les unes avec les autres pour obtenir le résultat souhaité. Nous allons, dans ce chapitre, voir quelques commandes de base, utilisables dans de très nombreux contextes.

Les commandes cat, head et tail

La commande **cat** est utilisée pour afficher le contenu d'un fichier, son usage initial est la concaténation de plusieurs fichiers. Ainsi, ce programme reçoit en paramètre une liste de fichiers qui seront affichés sur la sortie standard. Son usage est le suivant :

cat nomfichier [nomfichiers...]

exemple : cat fichier.txt	=>	affiche le contenu de fichier.txt a l'écran
exemple : cat debut.txt corps.txt fin.txt	=>	affiche à l'écran, à la suite, les trois fichiers (concaténation)
exemple : cat debut.txt fin.txt > tout.txt	=>	le fichier tout.txt contiendra la concaténation des deux fichiers : début et fin

cat illustre le fonctionnement de la plupart des commandes shell : le programme ne se préoccupe pas de savoir si la cible est un fichier, l'écran ou même un périphérique. Les programmes envoient leur résultat sur la sortie standard, c'est ensuite le shell, par le système de redirection qui se chargera des procédures d'écriture dans les fichiers ou périphériques. C'est une approche différente de la programmation classique où le développeur doit couramment écrire plusieurs fonctions de sorties en fonction du média utilisé. Par conséquent, bien que les commandes shell marchent pour la plupart avec des fichiers binaires, elles sont généralement mieux adaptées aux fichiers texte.

Nous venons de voir **cat** que l'on utilise pour afficher l'intégralité d'un fichier. Dans de nombreux cas, vous ne serez intéressés que par les premières ou dernières lignes d'un fichier : pour traiter une entête ou vérifier le format des données par exemple ou plus couramment pour afficher les dernières lignes de la trace de l'un de vos programmes ou site web... Les commandes à utiliser sont alors **head** pour l'affichage des premières lignes et **tail** pour les dernières lignes. Ces commandes s'utilisent avec des options permettant de fixer le nombre de lignes/mots/octets... à envoyer sur la sortie standard. Voici quelques exemples d'utilisation :

tail -100 fichier.txt	=>	affiche les 100 dernières lignes de fichier.txt

head -100 fichier.txt	=>	affiche les 100 premières lignes de fichier.txt
tail -100 fichier.txt > fin.txt	=>	redirige dans le fichier fin.txt les 100 dernières lignes de fichier.txt

Voyons un exemple un peu plus élaboré :

tail -10 fichier.txt | head -1

Ici nous utilisons le « tube » vu dans les précédents chapitres pour passer à la commande **head** le résultat de la commande **tail** précédente. Nous pouvons alors noter que les commandes **head** et **tail** ne prennent pas forcement de nom de fichiers en paramètre, dans ce cas, les données d'entrée sont lues sur l'entrée standard. Vous pouvez expérimenter cela à tapant la commande « **tail -5** », tapez alors une dizaine de ligne puis fermez l'entrée standard en appuyant simultanément sur les touches CTRL et d (CTRL+d), vous verrez alors apparaitre les cinq dernières lignes tapées à l'écran.

Bref, revenons à cet exemple, la première commande envoie sur sa sortie les dix dernières lignes de fichier.txt, ces lignes arrivent, au travers du tube à l'entrée de la seconde commande head qui ne va conserver que la première des lignes. Nous venons de créer une nouvelle commande par la combinaison de deux : la commande qui extrait la *nième* ligne en partant de la fin.

Les commandes less et more

Les commande **less** et **more** ne sont pas vraiment utilisables dans des scripts, elle rendent par contre beaucoup plus simple la vie à l'utilisateur : lorsque l'on souhaite visualiser le contenu d'un fichier, avec **cat** par exemple, tout le fichier sera affiché à l'écran, d'une traite. A moins que vous ne décodiez la matrice en temps réel, je ne pense pas que vous puissiez lire grand chose... Les fonctions d'historique des terminaux étant assez limitées (surtout au début), des commandes plus adaptées au besoin de l'utilisateur ont été créées. La première, **more** permet un affichage page à page du contenu d'un fichier, voire de n'importe quel flux de caractères envoyés sur l'entrée de **more.** Le défilement s'arrêtera donc dès qu'une page complète sera affichée. L'appuie sur la touche espace permettra d'afficher une nouvelle page.

more permet de stopper le défilement mais n'est pas conçu pour remonter dans le flux, la commande **less** s'utilise alors en remplacement. Il est en effet possible de naviguer dans le flux de caractères transmis à **less** avec les flèches du clavier. Vous avez déjà croisé la commande **less** cachée derrière la commande **man** : la navigation dans le manuel est de type **less.** Pour quitter **less** ou **more,** il suffit d'utiliser la touche « **q** »

Exemples d'utilisation:

find / -name "*" | more	=>	affiche tous les fichiers du système, page par page
less fichier.txt	=>	affiche à l'écran le contenu de fichier.txt avec less, autorisant la navigation montante et descendante

La commande grep (version simplifiée)

La commande **grep** offre une autre façon de sélectionner des lignes, non pas sur le numéro de la ligne mais sur son contenu, c'est donc une commande que nous utiliserons très fréquemment. Nous verrons par la suite que cette commande nous offre bien d'autres perspectives pour la recherche de motifs au travers d'expressions régulières. En attendant, commençons par l'utiliser

assez simplement. La syntaxe de **grep** est la suivante :

grep motif [fichiers...]

Le motif représente ce qui est recherché, on placera ici le mot clef à trouver. Il est ensuite possible de mettre une liste de fichiers dans lesquels chercher ce motif, mais il est aussi possible de passer le contenu par l'entrée standard, au travers d'un tube comme vu précédemment. **grep** va parcourir toutes ces données et envoyer sur sa sortie toutes les lignes contenant le motif.

Voici quelques exemples d'utilisation :

grep class *.java	=>	affiche à l'écran toutes les lignes contenant le mot « class » lors du parcours du contenu des fichiers se terminant pas « .java »
ls -l | grep paul:	=>	la commande ls liste les fichiers du répertoire courant, le résultat est transmis à grep qui sélectionnera les lignes contenant la chaîne « paul: ». Cette commande devrait donc sélectionner au moins les fichiers appartenant à l'utilisateur paul

Les commandes cut et tr

La commande **cut** vient en complément des commandes **head, tail** et **grep** en permettant, non pas l'extraction de lignes, mais l'extraction de colonnes dans une ligne. Ainsi, en mixant **head, tail, grep** et **cut**, il est possible de trouver un élément particulier d'un fichier. Il existe de multiples façons de sélectionner une colonne, la commande **man cut** vous donnera tous les détails, nous allons en voir une en particulier :

cut -d '*delim'* -f *colNum [fichiers...]*

Cette commande permet d'envoyer sur la sortie les données trouvées dans la colonne « *colNum* » parmi les données issues de l'entrée standard ou de fichiers. La séparation entre deux colonnes est identifiée par un caractère appelé délimiteur, ici nommé « *delim* ».

Concrètement, cela peut donner :

cut -d ':' -f 1 /etc/passwd	=>	affiche les logins (première colonne du fichier passwd) de tous les utilisateurs du système. Le délimiteur utilisé dans le fichier passwd étant « : »

Voici une ligne extraite de /etc/passwd (pour mémoire et sur MAC)

www:*:70:70:World Wide Web server:/Library/WebServer:/usr/bin/false

La commande précédente retournera pour cette ligne « www ».

Maintenant la même commande utilisée pour afficher le nom de l'utilisateur (ici Worl wide Web Server) : **cat /etc/passwd | cut -d ':' -f 5 .** L'utilisation ici du pipe n'a pour seule vocation que l'exemple, l'utilisation de cat ou l'ajout du nom du fichier en argument revenant grosso modo au même.

Il existe un cas où cette méthode ne fonctionne pas directement : prenons l'exemple de ls -l qui pour des raisons d'alignement et de présentation à l'utilisateur sépare les champs par des espaces dont le nombre va varier en fonction des données affichées :

ls -l

-rw-r—r-- 1 paul paul 50 Aug 13 15:06 toto.c

-rw-r—r-- 1 julie julie 36 Aug 13 15:07 tata.c

Ici, un espace sépare « julie julie » mais deux espaces séparent « paul paul » pour que l'alignement des noms de groupe soit correct. Si bien que si l'on souhaite utiliser la commande **cut** pour identifier le groupe propriétaire des fichiers, le champ dans la première ligne sera le 5e alors qu'il sera le 4e dans la seconde ligne.

La solution à ce type de problèmes passe par l'utilisation d'une autre commande : **tr** pour ***transform***, cette commande permet d'effectuer des modifications sur l'ensemble d'une chaîne de caractères. Elle peut être utilisée pour passer la chaîne en majuscule par exemple ou, comme nous allons le faire, pour supprimer des espaces en double. Encore une fois, c'est la commande **man tr** qui vous permettra d'en percer tous les secrets.

Pour notre part, nous l'utiliserons ainsi : tr -s ' ' pour supprimer tous les espaces. Si on applique cela à ls -l, le résultat est le suivant:

ls -l | tr -s ' '

-rw-r—r-- 1 paul paul 50 Aug 13 15:06 toto.c

-rw-r—r-- 1 julie julie 36 Aug 13 15:07 tata.c

Il nous est alors possible d'utiliser la commande **cut** vu précédemment :

ls -l | tr -s ' ' | cut -d ' ' -f 4	=>	pour obtenir le groupe de tous les fichiers
ls -l | tr -s ' ' | cut -d ' ' -f 4,5,9	=>	pour obtenir le groupe, la taille et le nom de tous les fichiers

Vous noterez qu'il est possible d'enchaîner autant de « tubes » que l'on souhaite dans une ligne de commande.

Enfin, une autre façon courante d'utiliser **cut** se fait avec l'option -b qui permet d'indiquer quels octets (comprendre caractères ici) l'on souhaite extraire. Dans le cas de ls -l, cet usage est utile pour extraire certains droits puisqu'il n'y a pas de séparateur entre les différents droits du fichier. Ainsi pour extraire les droits associés à *other (3e groupe de trois lettres commençant à l'index 8),* on pourra utiliser :

ls -l | cut -b 8-10

Les commandes column, join et paste

L'usage de column, join et paste est plus rare, mais ces commandes peuvent être fortement utiles. **column** est utilisé pour transformer une entrée en ligne en une sortie en colonne, ainsi, si l'entrée de la fonction est :

toto.c

titi.c

tata.c

La sortie sera :

toto.c titi.c tata.c

Cette fonction pourra être utile dans l'utilisation d'une boucle *for* par exemple où une liste de valeurs que doit prendre la variable est à donner en colonne. Nous verrons cela par la suite.

La commande **join** est un peu plus complexe : elle permet de fusionner des fichiers à partir d'une clef commune. **join** va parcourir les deux fichiers ligne à ligne et rechercher une clef présente dans les deux fichiers. La sortie sera la fusion des deux fichiers lorsque la clef est trouvée:

Si l'on a par exemple :

Fichier1	Fichier2	=> **column Fichier1 Fichier2** =>	Sortie
key1 a	key1 b		key1 a b
key2 c	key2 d		key2 c d
key3 y	keyA z		

=> les lignes 1 et 2 des fichiers sont fusionnées dans la sortie. Les lignes 3 n'ont pas de correspondance et sont donc rejetées.

La commande **paste** en est une version plus simple : elle réalise la fusion des lignes de deux fichiers, mais sans se soucier de quelconques clefs.

Fichier1	Fichier2	=> **paste Fichier1 Fichier2** =>	Sortie
a	b		a b
c	d		c d
y	z		y z

La commande sort

La commande **sort** permet de trier les lignes d'un fichier ou d'un flux reçues sur l'entrée standard. **sort** peut s'utiliser tout simplement de la sorte :

ls | sort => trie le résultat de la commande **ls** par ordre alphabétique.

De nombreuses options existent, réfèrerez-vous, comme toujours, au manuel (**man sort**) pour plus de détails.

Lecture de données sur l'entrée standard

Nous avons couramment besoin de lire des données sur l'entrée standard, il peut s'agir de saisies utilisateur ou de données issues de fichiers. Dans tous les cas, la lecture de données se fait par l'usage de la commande ***read***.

Utilisée directement, cette commande stockera le résultat de la lecture de l'entrée standard dans la variable REPLY (en majuscule). Ainsi nous pouvons écrire les lignes suivantes :

```
echo -n "Quel est votre prénom ?"
read
echo "Bonjour $REPLY !"
```

Il est aussi possible de l'utiliser en précisant une liste de variables, dans ce cas, les mots seront associée, un à un, dans les variables indiquées, la dernière variable recevra toute la saisie restante:

```
echo -n "Saisissez vos nom et prénoms"
read nom prenoms
echo "Votre nom est ${nom}, vos prénoms sont ${prenoms}
```

Si l'utilisateur saisit : Pinault paul jean nicolas ; alors la variable nom prendra comme valeur le premier mot, ici « Pinault » et prénom les mots restant « paul jean nicolas ».

Les processus

Avant tout... qu'est-ce qu'un processus ? C'est tout simplement un programme en cours d'exécution. Un programme est une suite d'instructions stockées, généralement sur le disque dur. Une fois lancé, il devient un processus : en plus des instructions contenues dans le programme, le processus contient des données et un contexte d'exécution.

A quoi cela peut-il servir de connaître le maniement des processus ? Pour répondre à cette question, je vais vous raconter ce qui m'occupe une partie de mon temps en ce moment et ce que la connaissance des processus m'apporte dans cette tâche. Nous utilisons dans mon entreprise un progiciel du marché dans une configuration très consommatrice de mémoire (entre 13 et 16 Go de mémoire), du fait, entre autre, de serveurs plutôt sous-taillés, cette application nous pose des problèmes dans plusieurs zones (Europe et USA), les erreurs qu'elle nous renvoie sont différentes dans ces zones et plutôt aléatoires. La question était donc de trouver une solution à ce souci et ce par deux moyens différents : d'une part démontrer le manque de mémoire des serveurs pour corriger le problème de fond et d'autre part trouver une solution pour que l'application puisse être redémarrée, y compris après un plantage sévère où la procédure classique d'arrêt ne rend même plus la main. Pourquoi vous parler de ceci ? Pour vous illustrer que ce type de question n'est pas posée à une équipe de développement, ni même à des spécialistes d'Unix, elle peut se poser à tout le monde, y compris au chef de projet. Mettre en évidence la consommation mémoire de l'application et la saturation du serveur se fait à l'aide de commandes d'analyse des processus telles que **ps** que nous verrons dans ce chapitre. Quant à la possibilité d'arrêter une application récalcitrante, c'est par l'emploi de signaux « tueurs » intégrés correctement dans le script d'arrêt que la solution a été apportée. Maintenant l'application tourne et surtout s'arrête quand on le lui demande. C'est par la connaissance des processus que la solution à ce problème a pu rapidement être apportée. Les message d'erreurs nous conduisant sur une piste différente mais erronée, conséquence de la surcharge du serveur mais non cause de l'incident.

Définition d'un processus

Un processus est donc un programme en cours d'exécution, il est défini par un ensemble de critères comme un identifiant (appelé PID pour Process IDentifier), le nom de l'utilisateur l'ayant lancé (ou utilisateur propriétaire), une quantité de mémoire utilisée, un état, un temps d'exécution...

Ces différentes informations attachées au processus sont consultables au travers d'une commande : **ps**. Il existe de nombreuses façon d'utiliser **ps** que nous verrons très bientôt.

Un processus c'est avant tout un programme, c'est à dire une suite d'instructions à exécuter. Ce programme est chargé en mémoire, il peut éventuellement être partagé entre plusieurs processus lorsque ceux-ci correspondent à plusieurs exécutions d'un même programme.

Un processus comprend une zone de données, dans cette zone sont stockées toutes les données relatives à l'exécution en cours, ce sont des données dynamiques, c'est à dire les tableaux, variables

que vous déclarez dans vos programmes.

Un processus comprend enfin un contexte d'exécution. Ce contexte est un ensemble de données relatives au processus et à la gestion, en générale, des processus. On y retrouvera les informations telles que le point d'exécution courant de l'application, la liste des zones mémoires occupées, l'heure de lancement, la commande tapée... Ce contexte est géré non pas par le processus mais par le noyau.

Une histoire de famille

Tout processus a un père, hormis le premier de tous, bien entendu. Celui-ci s'appelle, sous Linux, *init.* Le père sera le processus qui aura lancé le fils. Donc, lorsque vous lancez la commande **find** dans un shell, par exemple, vous créez un processus **find**, fils de **bash**.

La généalogie des processus est consultable à l'aide de la commande **pstree** qui s'utilise sans option particulière et affiche les processus sous forme d'un arbre.

Un processus hérite des propriétés de son père, ainsi il utilisera les entrées et sorties auxquelles est connecté le père. Cela signifie que si une redirection des entrées sorties est faite sur le père, elle s'appliquera de la même sorte aux fils. Cet héritage s'applique à toutes les propriétés du père : utilisateur ayant engendré le processus, la priorité d'exécution...

Cette histoire de famille serait des plus banales s'il n'était pas nécessaire que le père survive au fils sans quoi le fils deviendra ce que l'on appelle un zombi. Tout processus se voit contraint de terminer son existence par la transmission d'un code de retour à son père ; si celui-ci n'est à ce moment pas à même de recevoir ce message, le fils ne pourra disparaître et il restera sur le système son emprunte fantomatique, persistante jusqu'au retour du père. Pour simuler une telle situation, il suffira par exemple de stopper l'exécution du père à l'aide d'un signal SIGSTOP (nous verrons cela par la suite), le fils, une fois terminé passera alors dans l'état zombi (aussi noté defunc) en attendant que le père soit de nouveau actif. La terminaison du père ou sa relance (SIGCONT) permettront de faire disparaître le zombi. Attention, il ne sert à rien de s'acharner à tuer un zombi pour le faire disparaître : en étant déjà mort, il ne risque plus rien. C'est donc toujours le père du zombi qu'il faut retrouver pour le faire disparaître.

Le décès du processus père conduit à l'adoption des processus fils par le processus « init ». Comme tout processus doit avoir un père, au décès du père c'est donc « init » qui prendra soin de tous les rejetons orphelins. Lors de la fermeture du terminal, tous les processus lancés sur le terminal seront tués. Pour éviter cela, il faudra lancer les programmes en utilisant la commande « nohup » suivie de la commande à exécuter. A la fermeture du terminal, les processus lancés par « nohup » ne seront pas terminés, mais adoptés par « init ».

Le partage du temps de travail

Au royaume des processus, il n'existe pas du travail pour tout le monde en même temps : les unités de traitement (les CPU) sont de très loin moins nombreuses que le nombre de processus que le système doit traiter en parallèle. Pour illustration, la machine sur laquelle je tape ce texte fait actuellement tourner 143 processus différents (***ps aux | wc -l)*** alors que mon processeur n'a qu'une seule et unique unité de traitement. C'est donc par le partage de ce processeur en petites unités de temps que le système va traiter tous les besoins et me donner l'impression que tout fonctionne en parallèle. Disons que dans l'entreprise de mon ordinateur, il y a tout de suite 143 équipes qui se relaient toutes les 5 à 10 millisecondes, les unes à la suite des autres.

La gestion de la ressource processeur est sous le contrôle d'un programme particulier, au coeur du noyau : l'ordonnanceur. C'est ce programme qui décidera de quel processus pourra s'exécuter à un

moment donné. Son rôle est assez compliqué, car il doit gérer des processus plus pressés que d'autres (notion de priorités entre processus) ; il doit gérer des processus en attente de données et donc mis au repos forcé attendant que l'utilisateur veuille bien presser une touche du clavier, ou que cet escargot de disque dur nous livre enfin le fichier tant attendu... et oui, avec des journées de 10 millisecondes, le travailleur processus attend l'équivalent de mois entre chaque pression de mon clavier !

L'ordonnanceur distingue donc différents états que peuvent prendre chacun des processus :

- L'état élu, tout d'abord, est l'état d'un processus en cours d'exécution.
- L'état bloqué est attribué au processus en attente d'un évènement externe (comme une pression du clavier, des données issues du disque dur...) ; un processus bloqué devra repasser par l'état prêt avant d'être à nouveau élus, une fois son tour venu.
- L'état prêt, indiquant que le processus pourra à nouveau être exécuté, dès lors que ce sera son tour.

A chaque cycle d'exécution (toutes les fameuses 10ms), l'ordonnanceur devra choisir parmi tous les prêts, le ou les élus qui auront le privilège de s'exécuter sur les unités de traitement. Ce choix se fait sur deux critères : la priorité d'une part et le temps d'attente d'autre part. L'idée étant simplement que plus un processus attend et plus il a de chance d'être élu, plus il est prioritaire et plus il a de chance d'être élu. La priorité est aussi appelée « gentillesse » ou « nice » en anglais.

L'ordonnanceur est dit préemptif, c'est à dire qu'il a le pouvoir d'arrêter le processus en cours d'exécution et ainsi donner la main à un autre. Il existe des systèmes coopératifs, alors c'est le processus lui même qui doit avoir la gentillesse de s'arrêter et de laisser la main à l'ordonnanceur qui désignera un successeur. Ce dernier cas est un peu l'histoire de la radio RDS et de l'interruption des programmes pour un message d'urgence : souvent on vous rend la main et vous retournez à votre programme initial et puis des fois, vous restez bloqué sur autoroute-info car le retour n'a pas fonctionné... Dans un système préemptif, la main ne sera pas rendu, mais reprise par l'ordonnanceur.

On distingue en outre deux politiques d'ordonnancement, la première est dite à « temps partagé », c'est celle que nous venons de décrire et c'est aussi la plus courante sur les ordinateurs personnels et serveurs classiques. Elle donne l'illusion à chaque traitement de s'exécuter en parallèle de façon équitable. La seconde politique est dite « temps réel », celle-ci est courante dans l'informatique embarquée et industrielle, elle garantit des contraintes temporelles. Elle garantit par exemple un temps de réponse, ou plus précisément qu'il ne s'écoulera pas plus d'un laps de temps déterminé avant que le processus ne soit de nouveau élu. Une illustration imagée de ceci est l'utilisation d'un ordonnanceur dans la gestion d'un ABS : il vaut mieux être sûr que le programme de gestion du freinage ne mette pas trop de temps avant d'activer les plaquettes ; pour ce qui est d'éteindre ou allumer les clignotants... on verra un peu plus tard !

Autopsie d'un processus

L'accès aux informations d'un fichier se fait par l'usage de la commande *ls*, comme nous l'avons vu, pour ce qui est des processus, la commande à utiliser est ***ps***. Nous avons déjà abordé le fait que les processus sont décrits dans un répertoire spécialisé ***/proc*** ; ce répertoire contient un répertoire pour chacun des processus du système, ce répertoire contient de très nombreuses informations sur chacun des processus. Ces informations sont sans nul doute les plus complètes mais ce répertoire n'est pas vraiment orienté « utilisateur final » et les informations qui y sont contenues sont loin d'être un standard. Bref, la commande ***ps*** est donc **LA** bonne façon d'accéder aux informations des processus.

ps sans arguments affichera la liste des processus courants :

```
paul@chaton:~> ps
```

```
  PID TTY          TIME CMD
 6021 pts/1    00:00:00 bash
 6029 pts/1    00:00:00 ps
```

La commande affiche donc simplement un minimum d'informations sur le shell en cours et les processus lancés depuis ce shell, à savoir la commande ***ps*** elle-même. Ces informations sont assez limitées mais permettent un début d'explication. Chaque processus est identifié par un *PID* ; il s'agit du Process IDentifier. C'est un numéro unique affecté à chaque processus lors de sa création, il servira à identifier le processus de façon formelle pour, par exemple, communiquer avec lui.

ps s'utilise généralement adjoint à d'autres options permettant de lister tous les processus, il existe de multiples combinaisons d'options, mais les plus courantes sont les suivantes :

- ***ps aux***

```
paul@chaton:~> ps aux
USER       PID  %CPU %MEM    VSZ   RSS   TTY      STAT START   TIME COMMAND
root         1   0.0  0.0    744    72    ?        Ss   Sep02   0:01 init [5]
root         2   0.0  0.0      0     0    ?        S    Sep02   0:00 [migration/0]
root         3   0.0  0.0      0     0    ?        SN   Sep02   1:37 [ksoftirqd/0]
root      3289   0.0  0.0   1824   564    ?        S    Sep02   0:54 hald-addon-storage: polling /dev/sda
root      3291   0.0  0.0   1828   564    ?        S    Sep02   0:54 hald-addon-storage: polling /dev/sdb
root      3293   0.0  0.0   9952   468    ?        S<sl Sep02   0:00 /sbin/auditd -n
root      3294   0.0  0.0   1828   564    ?        S    Sep02   0:53 hald-addon-storage: polling /dev/sdc
nobody    3330   0.0  0.0   1628   388    ?        Ss   Sep02   0:00 /sbin/portmap
root      3367   0.0  0.2  11204  1876    ?        S    Sep02   0:14 /usr/sbin/snmpd -r -A -LF d /var/log/net-s
ntp       3731   0.0  0.1   4548  1068    ?        Ss   Sep02   0:10 /usr/sbin/ntpd -p /var/lib/ntp/var/run/ntp/n
root      3842   0.0  0.0   2056   556    tty4     Ss+  Sep02   0:00 /sbin/mingetty tty4
root      6037   0.0  0.1   4268  1440    tty1     Ss+  Sep02   0:01 -bash
root      7307   0.0  0.0   2416   568    ?        Ss   Sep02   0:00 /sbin/syslog-ng
root      7310   0.0  0.0   1724   508    ?        Ss   Sep02   0:00 /sbin/klogd -c 1 -x -x
root      7358   0.0  0.0   3380   660    ?        Ss   Sep02   0:00 /opt/kde3/bin/kdm
paul      7457   0.0  0.0   3724   404    ?        Ss   Sep02   0:00 /usr/bin/dbus-daemon --fork --print-pid 4 --print-ad
```

Cette combinaison d'options permet d'afficher tous les processus du système (ici une sélection seulement) en présentant les informations suivantes :

- USER : le propriétaire du processus (qui l'a lancé).
- PID : identifiant du processus.
- %CPU : la part du processeur utilisée par ce processus.
- %MEM : la part de mémoire utilisée par ce processus.
- VSZ : la taille de la mémoire virtuelle allouée au processus en Ko, incluant la mémoire RAM (physique) utilisée et la mémoire logique (swap).
- RSS : la taille allouée dans le swap, en Ko.
- TTY : indique le périphérique terminal depuis lequel ce processus a été lancé.
- STAT : état du processus (S=en attente d'un évènement, R=en cours ou pret, T=arrété...)

- START TIME : heure de départ du processus.
- COMMAND : ligne de commande associée à ce processus.

- ***ps ef***

```
paul@chaton:~> ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Sep02 ?        00:00:01 init [5]
root         2     1  0 Sep02 ?        00:00:00 [migration/0]
root         3     1  0 Sep02 ?        00:01:37 [ksoftirqd/0]
```

Cette combinaison d'option permet d'afficher tous les processus du système (ici une sélection seulement) en présentant les informations suivantes :

- UID : Le propriétaire du processus (qui l'a lancé)
- PID : identifiant du processus.
- PPID : identifiant du père du processus (quel processus a lancé celui-ci).
- C : taux d'utilisation du CPU moyen depuis son lancement.
- STIME : heure de départ du processus.
- TTY : indique le périphérique terminal depuis lequel ce processus a été lancé.
- TIME : temps cumulé d'utilisation du processeur par le processus (ici les processus sont lancés depuis presque un mois, mais ils n'ont utilisé que quelques secondes du processeur).
- CMD : ligne de commande associée à ce processus.

Il existe bien sûr d'autres combinaisons d'options permettant d'affiner l'utilisation mémoire, l'utilisation cpu... filtrer les processus d'un utilisateur (***ps -fu user***)...

ps est un outil basique mais puissant pour analyser l'état d'un système et ainsi déterminer l'utilisation mémoire d'un programme, vérifier le sizing du système... C'est donc à la fois l'outil du développeur pour mieux comprendre le fonctionnement de son programme, celui de l'exploitant pour comprendre les problèmes, celui du chef de projet pour justifier ses demandes d'augmentations mémoires ou processeurs.

D'autres commandes permettent d'analyser le système et plus particulièrement les processus. La commande ***top*** permet d'avoir un affichage temps réel des processus en cours d'exécution qui seront triés selon différents critères. Sur les systèmes AIX, cette commande sera ***topas*** fournissant d'autres informations complémentaires. La commande ***vmstat*** est aussi très répandue et donne des statistiques globales sur l'utilisation du serveur.

En outre la commande ***ps*** peut aussi être utilisée dans des scripts pour identifier des processus et communiquer avec eux de façon automatisée.

Exécution d'un processus

Un processus est l'exécution d'un programme, il se lance donc simplement en tapant son nom :

paul@linux:~> ***ls***

est la création d'un processus ***ls*** par exemple.

Un processus peut toutefois être exécuté de plusieurs façons différentes :

- ls **&** : lancement du processus en arrière plan.

  Ce mode de lancement permet à l'utilisateur de reprendre la main directement après le lancement du processus plutôt qu'à la fin de celui-ci.

- **nohup** ls **&** : lancement du processus sans le rattacher au terminal courant.

  De cette façon, le processus continuera son exécution, y compris lorsque l'utilisateur fermera le terminal. De façon pratique cette option permet de fermer sa session sur un serveur sans pour autant arrêter le processus ainsi lancé.

- ls **;** ps : lancement séquentiel de processus.

  L'utilisation du séparateur « ; » permet d'exécuter des programmes en séquence (l'un après l'autre).

- ls -l | grep : lancement de processus en parallèle.

  Par opposition au séparateur précédent, comme nous l'avons vu, l'utilisation du « | » (tube) permet d'exécuter des processus en parallèle. Ceux-ci seront toutefois dépendants des uns des autres puisque leurs entrées et sorties sont chaînées.

- cd /etc **&&** ls : lancement conditionné.

  L'utilisation du séparateur « && » permet de conditionner l'exécution du second programme au bon déroulement du premier. Dans cet exemple, *ls* ne sera exécuté que si la commande *cd* se déroule bien.

- cd /toto **||** echo "/toto n'existe pas" : lancement conditionné.

  L'utilisation du séparateur « || » permet de conditionner l'exécution du second programme par l'échec du premier. Dans cet exemple, *echo* affichera son message si la commande *cd* a échoué.

- (cd /toto || echo "/toto n'existe pas" >&2) > resultat.txt

  L'utilisation de « ( ) » permet de regrouper des commandes, ainsi dans l'exemple, les sorties standards des commandes cd et echo seront redirigées vers le fichier resultat.txt.

Pour déterminer si un processus s'est ou non bien exécuté, chaque processus retourne un code, ce code est un entier compris entre 0 et 127. La valeur 0 indiquera, par convention, que le programme s'est déroulé normalement. Toutes autres valeurs correspondront à un message d'erreur. Le choix de l'affectation entre erreur et retour est laissé à la discrétion de chaque programme. Il n'existe pas de conventions là-dessus.

L'utilisation des codes de retour est primordiale, lors de l'enchaînement de programmes dans une application batch (dans la réalité), c'est ce code qui permettra de déterminer si l'on peut exécuter les programmes suivants ou si l'on devra stopper l'exécution, générer des alertes... et donc réveiller des personnels d'astreintes à 3 heures du matin ou non...

Le code de retour d'un processus est forcé par l'utilisation de la commande ***exit*** suivie du code à retourner. Dans le cas d'un regroupement (utilisation de « ( ) ») le code du dernier processus exécuté sera retourné.

Le code de retour est lu, juste après le retour du processus, dans la variable « **?** » ; on l'affichera donc de la façon suivante : ***echo $?***

Voici un exemple plus complet basé sur ce que nous avons vu :

paul@linux> (cd /tmp 2> /dev/null || (echo "repertoire inexistant" >&2 ; exit 1)) && ls

Cette commande va commencer par exécuter la commande ***cd***.

- si celle-ci n'aboutit pas, la commande ***echo*** sera appelée puis la commande **exit**. L'appel à exit avec comme valeur 1 va indiquer que la commande s'est finie en erreur. De ce fait, l'enchaînement au travers de « && » ne se fera pas.
- si celle-ci aboutit, la commande retourne 0 et l'enchaînement par « || » ne se fera pas. Le premier groupe étant terminé en succès, l'enchaînement « && » pourra se faire et la commande ***ls*** sera exécutée.

Un code de retour particulier : 137 sera utilisé pour indiquer qu'il n'y a pas de code de retour. Cela arrive par exemple lorsque le processus est tué avant qu'il ne retourne une valeur.

D'autres variables permettent d'accéder à différentes informations sur les processus :

- Il est ainsi possible de connaître le PID du processus en cours en interrogeant la variable « $ » (par exemple par « echo $$ »).
- Il est aussi possible de connaître le PID du dernier processus lancé en arrière plan en consultant la variable « ! », par exemple par « echo $! ».

Communiquons avec les processus

Les processus utilisent un système de communication basé sur l'échange de signaux. Un signal est simplement un évènement qui est transmis au processus, il n'est porteur d'aucune donnée. Un signal a une signification préétablie, dans la majeure partie des cas. Cette signification conduit à une action par défaut du processus qui dans bon nombres de cas se traduit par la terminaison prématurée du processus. Il est cependant possible de redéfinir le comportement de certains signaux. Il existe aussi 2 signaux sans comportement pré-établit, ceux-ci pourront alors être définit, en toute liberté, par l'utilisateur.

Détourner un signal servira par exemple à sauver les données en cours de traitement avant l'arrêt forcé d'un processus. Les signaux utilisateurs pourront être utilisés pour commander l'ouverture d'une interface graphique de configuration d'un démon par exemple.

L'émission d'un signal vers un processus se fait par l'usage de la commande « **kill** », cette commande, bien que signifiant « tuer » en anglais n'a pas pour vocation unique l'envoi de signaux mortuaires aux processus, elle permet de transmettre n'importe quel signal à un processus. Les signaux sont émis en utilisant leur nom ainsi que le PID du processus. Il faudra donc souvent avoir recours à la commande **« ps** » pour déterminer le PID du processus cible avant de communiquer avec lui.

Outre les signaux que l'utilisateur peut envoyer à un processus, (ou ceux qu'un processus peut envoyer à un autre processus), le système transmettra lui aussi des signaux aux processus, bon nombre de ceux-ci devraient d'ailleurs vous évoquer des messages d'erreurs qui ne vous seront pas inconnus.

Commençons par les signaux systèmes remontant des composants matériels, ceux-ci sont envoyés lorsque qu'une opération impossible à réaliser ou interdite est demandée par le processus. En réponse, celui-ci sera généralement tué et sa mort conduira à la génération d'un fichier « core » (core dump). Un fichier « core » correspond à l'écriture sur le disque d'une image de la mémoire

utilisée par le processus au moment de son arrêt. Ce fichier peut être utilisé pour rechercher l'erreur qui a conduit à l'arrêt du programme.

Le signal SIGILL sera émis lorsque le processeur recevra une instruction illégale à exécuter ; ce signal arrive généralement lorsque le processeur se retrouve à exécuter les données d'un programme plutôt que ses fonctions, ce signal fait souvent suite à un débordement de pile. Il indique en tout cas que la poursuite de l'exécution en cours est impossible puisque l'instruction courante souhaité est inconnue du processeur.

Le signal SIGFPE est émis par le composant de calcul en virgule flottante lorsque le calcul ne peut être effectué, comme lors d'une division par zéro par exemple. Le programme devra être arrêté car les données ne seront plus cohérentes.

Le signal SIGSEGV sera sans doute l'un des plus courants que vous allez rencontrer, il indique que votre programme a tenté d'accéder à une zone mémoire non autorisée. Concrètement, cela signifie par exemple que vous avez déclaré un tableau de 10 éléments et que vous essayez d'accéder, par exemple, au 11^{e}. Ce signal indique que votre programme n'est pas correct et vos données n'étant plus pertinentes, il sera arrêté.

Les signaux suivants sont émis par le système d'exploitation lors de la mort de certains processus :

Nous avons déjà abordé le cas du signal SIGPIPE, il indique, lors de l'utilisation d'un tube que le lecteur s'est terminé avant l'écrivain ; le rôle de l'écrivain n'a donc plus de sens, il doit être arrêté.

Le signal SIGHUP est émis lorsque le terminal est fermé à tous les processus rattachés à ce terminal. C'est ainsi que tous vos processus en cours, lancés depuis un terminal, sont fermés lorsque vous fermez la fenêtre du bash.

Le signal SIGCHLD est émis vers le père chaque fois qu'un de ses fils se termine. Ce signal est sans effet particulier, il pourra être redéfini selon vos besoins.

D'autres signaux sont plutôt émis par l'utilisateur ou d'autres processus, ils permettent le contrôle de l'état du processus :

Le signal SIGSTOP permet de stopper le processus en cours. Il s'agit bien de stopper et non d'arrêter le processus en cours, c'est à dire que ce signal ne termine pas le processus mais le gèle ; c'est à dire que le processus restera à l'état bloqué et ne pourra être réélu tant qu'il sera dans cet état. Ce signal pourra être émis au processus en cours, depuis le clavier en tapant les touches CTRL+Z.

Le signal SIGCONT permet de reprendre l'exécution d'un processus précédemment stoppé, c'est à dire que le processus repasse de l'état bloqué à l'état prêt.

Le signal SIGTERM permet de demander l'arrêt du processus. Ce signal peut être redirigé par le programmeur pour effectuer par exemple une sauvegarde préalable, ou simplement pour empêcher ce type d'arrêt.

Le signal SIGINT permet de demander l'interruption d'un processus et donc de terminer le processus. Ce signal peut être émis par l'utilisateur, au clavier, pour le processus en cours en appuyant sur les touches CTRL+C.

Le signal SIGKILL, le plus connu, demande l'arrêt forcé du processus. Ce signal ne peut être détourné, quoi qu'il arrive, à réception de ce signal, le processus doit se terminer.

Il reste enfin quelques signaux utiles :

Le signal SIGALRM est transmis par la commande « **alarm** », cette commande lance un timer et émet ce signal après un temps souhaité par l'utilisateur. Ce signal sera utilisé par exemple pour répéter une action à fréquence régulière ou après un temps donné. Par exemple, on pourra demander une saisie utilisateur puis l'arrêter automatiquement pour choisir une valeur par défaut si l'utilisateur

ne saisit rien.

Les signaux SIGUSR1 et SIGUSR2 sont des signaux qui peuvent être définis par l'utilisateur, ils n'ont donc pas de comportement par défaut.

Il existe d'autres signaux moins courants dont vous trouverez les détails dans les manuels de la commande **« kill »** ou des signaux : **« man 7 signal »**.

Les signaux sont transmis par la commande **kill**, soit en utilisant le nom du signal comme nous venons de le donner, soit en utilisant un chiffre associé à ce nom. Par exemple :

paul@linux:~> kill -SIGKILL 12345

paul@linux:~> kill -9 12345

Les deux commandes envoient le signal KILL au processus 12345.

Pour intercepter un signal, nous utiliserons la commande « **trap** » ; bien que nous n'ayons pas encore vu les scripts, regardons la suite de commande suivante :

```
#!/bin/bash

trap 'echo "Reception du signal INT" ' SIGINT
trap 'echo "Reception du signal QUIT" ' SIGQUIT
while [ 0 ]; do  sleep 1 ; done
```

Ce programme est une boucle infinie, qui intercepte les signaux INT et QUIT. A réception de l'un de ces signaux, son comportement par défaut est détourné et le message correspondant apparaît. Ce script détourne SIGINT, ainsi, l'action des touches CTRL+C ne terminera plus le programme et affichera seulement le message associé. Le processus pourra, bien sûr, être terminé par l'emploi d'un signal plus radical comme SIGKILL par exemple, qui ne peut être détourné.

Les expressions rationnelles

Nous avons précédemment abordé les Jokers du shell permettant avec quelques caractères particulier de représenter des patterns pour identifier des fichiers. Les expressions rationnelles sont un outil encore plus puissant permettant de représenter des chaînes de caractères quelconques. Je n'ai aucun doute sur le fait que les expressions rationnelles puissent devenir vos meilleures amies dans votre vie professionnelle tellement elles peuvent vous simplifier la vie lors de rechercher-remplacer fastidieux par exemple.

C'est un en effet un outil couramment utilisé par les informaticiens pour réaliser rapidement des opérations de masse sur des fichiers, comme ajouter par exemple une colonne nécessaire à une nouvelle version de programme. C'est aussi l'outil qui vous permettra de reformater rapidement un fichier pour lequel votre programme, buggé, aurait oublié quelques données... Les expressions rationnelles ont l'intérêt d'être un langage universel, si bien qu'elles sont intégrées dans de nombreux éditeurs de texte (vi, eclipse, pspad, ultra-edit...) mais aussi dans la majeure partie des langages de programmation (C, Java...). Il s'agit donc d'un outil très pratique pour le développeur : elles permettent, comme nous le verrons de vérifier simplement une saisie utilisateur, quelle qu'elle soit, de déchiffrer un flux, comme de l'xml par exemple... Les applications sont donc très nombreuses et savoir les utiliser vous fera gagner beaucoup d'heures de code, car les reproduire par programmation devient vite fastidieux.

En Unix, les expressions rationnelles sont principalement associées aux commandes **grep** et **sed**. Elles permettent de rechercher un motif pour l'afficher, le remplacer, vérifier une syntaxe...

Les atomes

Nous avons utilisé les expressions rationnelles dans leur forme la plus banale en recherchant les lignes de */etc/passwd* contenant la chaîne *root* par exemple :

```
paul@linux:~> egrep root /etc/passwd
```

La suite de caractères *root* est ici une expression rationnelle, elle signifie que dans le fichier *passwd,* nous souhaitons identifier les lignes contenant, n'importe où, la suite de caractères *r o o t.* Pour chaque caractère, on parle d'atome : un atome peut être un caractère, mais ce peut aussi être un ensemble de caractères ou un caractère quelconque.

La notion d'ensemble est décrite comme en shell, en mettant entre crochets la liste des caractères possibles. L'utilisation d'alias d'ensemble étant, là aussi possible. Ainsi, il est possible de décrire que le caractère peut être choisi dans l'ensemble des voyelles de la façon suivante : « [aeiouy] ». Il est aussi possible d'indiquer une lettre minuscule à l'aide d'un alias : « [a-z] ». Cet ensemble peut être inversé en utilisant le caractère « ^ » ainsi l'ensemble des consonnes pourrait être déclaré de la façon suivante : [^aeiouy] (cet exemple étant faux puisqu'il autorise par exemple les chiffres).

La notion de caractère quelconque est représentée par le caractère « . » correspondant, en quelque sorte au « ? » du shell.

Ces notations permettront par exemple de rechercher, dans un ensemble de fichier texte, toutes les suites de caractères pouvant représenter une plaque d'immatriculation. En imaginant, pour simplifier qu'une plaque est une suite de 4 chiffres, suivie de 2 lettres et 2 chiffres, séparés par des espaces, l'expression rationnelle identifiant ces motifs sera la suivante :

```
[0-9][0-9][0-9][0-9] [A-Z][A-Z] [0-9][0-9]
```

Cette notation est donc assez simple, mais vous noterez que ainsi, elle est plutôt fastidieuse.

Les répétitions

Les répétitions s'appliquent aux atomes pour indiquer s'ils sont obligatoires et combien de fois on pourra les compter.

La répétition « ? » indique que l'atome qui le précède sera présent, une fois, ou non. Dans l'exemple précédent, comme le premier des nombres peut être composé de 1 à 4 chiffres, les 3 premiers sont facultatifs, l'expression rationnelle suivante sera donc plus juste :

```
[0-9]?[0-9]?[0-9]?[0-9] [A-Z][A-Z] [0-9][0-9]
```

La répétition concerne toujours l'atome le précédant, ainsi, le caractère « ? », seul, n'a aucune signification.

Il est possible d'indiquer une répétition quelconque avec le caractère « * ». Vous ferez attention à ce faux ami, puisque la description d'une suite quelconque de caractères devra donc s'écrire : « .* » Le « . » indiquant « caractère » et l' « * » indiquant « répété autant que nécessaire, voire absent ».

La répétition « + » indiquera la présence de l'atome au moins une fois ; il serait donc possible de réécrire l'expression que nous utilisons comme modèle de la façon suivante :

```
[0-9]+ [A-Z][A-Z] [0-9][0-9]
```

Toutefois cette écriture serait fausse car elle autorise les nombres de plus de 4 chiffres, ce qui ne correspondrait plus à une plaque. Il est donc nécessaire d'utiliser d'autres symboles de répétition : les ensembles de répétition. Ceux-ci sont définis entre {}.

L'utilisation de la répétition {2} par exemple, indiquera que 2 atomes consécutifs sont obligatoires. Il sera possible de définir des bornes minimum et maximum en utilisant un intervalle : {*min,max}*. Cette notation permet donc de redéfinir la notion de plaque de la façon suivante :

[0-9]{1,4} [A-Z]{2} [0-9]{2}

Les intervalles peuvent aussi être notés comme ouverts, « {*min*, } » signifiera « au moins *min* fois et « {, *max*} se traduira pas « au plus *max* fois ».

Les motifs à choix multiples et le bloc

Pour être plus complet sur notre notion de plaque d'immatriculation, il faudrait pouvoir indiquer que les départements peuvent être représentés soit par deux chiffres, soit par les chaînes 2A et 2B. Ce choix peut être explicité par une expression alternative indiquée par le symbole « | » utilisé avec les « ( ) ». Notre exemple sera écrit comme suit :

[0-9]{1,4} [A-Z]{2} ([0-9]{2}|2[AB])

Il est possible d'utiliser plusieurs alternatives dans une même parenthèse ou non.

Les « ( ) » définissent en réalité des blocs qui seront alors considérés comme des atomes, c'est à dire qu'il est possible d'associer une répétition à un bloc. Ainsi, nous pouvons, par exemple, identifier les nombres composés d'un nombre pair de chiffres de la façon suivante :

([0-9]{2})+

Cette expression indique que nous recherchons de répétitions de 2 chiffres quelconques. Ainsi seront identifiés des nombres comme 1223 ou 12 ou 125489 ; les nombres 123, 87914 seront ignorés.

Une forme plus évoluée de la répétition des blocs permet de demander le rappel exact d'un bloc. Dans ce cas, appliqué à l'exemple précédent, les nombres 1212 et 4848 pourront être identifiés alors que 1223 ne le sera pas car 12 et 23 ne sont pas identiques. L'expression sera écrite de la façon suivante : « ([0-9]{2})\1 ».

Cette répétition particulière se note « *n* » où *n* correspond au numéro du bloc, dans le cas où il y a plusieurs blocs de définis, ils s'appelleront (1,2,3...), les nombres étant associés de gauche à droite.

Un autre exemple

Nous pouvons rechercher la liste des décimaux relatif d'un programme, cette fonction permettra par exemple de rechercher toutes les constantes définies en dur dans le programme :

[+-]?([0-9]+|[0-9]+\.[0-9]+)

Cette expression a la signification suivante :

Un nombre décimal peut commencer par les symboles + ou -, mais ceux-ci sont facultatifs, ainsi on définira [+-]? un atome constitué de l'ensemble [+-] présente 0 ou 1 fois.

Un nombre décimal peut être un nombre entier, dans ce cas il se construit de la façon suivante : [0-9]+ . Il peut aussi être décimal, dans ce cas, il se note « [0-9]+\.[0-9]+ ». Vous remarquerez que le caractère « . » est ici précédé d'un « \ » servant à l'identifier comme le caractère « . » plutôt que comme l'atome « . » signifiant « n'importe quel caractère ».

Ces deux cas sont associés dans une alternative de la façon suivante :

([0-9]+| [0-9+]\.[0-9]+)

Si nous exécutions ce motif, nous nous rendrions compte qu'en réalité il n'est pas suffisant à lui même, en effet, comme rien ne précise que le motif ne peut pas être pris au beau milieu d'un mot, nous pourrions identifier le nom d'une variable du type *var2* par exemple.

Nous devons donc adjoindre au motif des informations de positionnement. Ici, nous souhaitons par exemple que notre nombre soit encadré de séparateurs comme « =, ,; ». Nous noterons l'exemple de la façon suivante :

[=;][+-]?([0-9]+|[0-9]+\.[0-9]+)[=;]

La localisation des motifs

D'une façon plus générale, il est souvent nécessaire de situer le motif dans la ligne, certains symboles particuliers permettent de spécifier que le motif soit en début de ligne ou fin de ligne, c'est le cas respectivement de « ^ » et « $ ». Il sera ainsi possible de décrire dans le fichier *passwd* un login comme étant la première suite de caractères avant le séparateur « : ». Ce motif s'écrira ainsi :

^([a-Z]|[0-9])+:

Localiser le motif que l'on cherche est aussi primordial pour le définir : cherchons par exemple à définir un mot, nous pourrions l'écrire comme étant une suite d'au moins un caractère minuscule, commençant éventuellement par une majuscule:

([A-Z][a-z]*|[a-z]+)

Cette définition peut sembler juste ; elle sera pourtant insuffisante car elle ne situe pas la chaîne de caractères, c'est à dire que si nous prenons « mot » par exemple, « m », « ot » ou « mo » correspondent au motif, or il ne s'agit pas de mots. Un mot n'est donc pas seulement défini comme étant une suite de caractères mais comme étant une suite de caractères alphabétiques entourée de caractères non alphabétiques. Ainsi la définition d'un mot sera plus juste ainsi :

[^A-Za-z]([A-Z][a-z]*|[a-z]+)[^A-Za-z]

Le remplacement de motifs

La dernière opération que nous verrons à propos des motifs concerne le remplacement de motifs. Vous vous êtes peut être demandé dans l'exemple précédent pourquoi identifier le login dans le fichier *passw* puisque toutes les lignes en contiennent un ? Il ne s'agirait pas de filtrer des lignes puisque ce cette façon, toutes seraient ramenées. Il s'agit alors de remplacer dans le fichier tous les login par autre chose ou simplement de les transformer. Les expressions régulières permettent cela de manière simple avec une commande tout aussi puissante que **grep** : « **sed** ».

La syntaxe d'un rechercher-remplacer est la suivante :

sed -e 's/expression_recherchée/expression_de_remplacement/'

L'expression recherchée est une expression régulière, l'expression de remplacement peut être une chaîne statique ou une chaîne composée d'éléments statiques et d'éléments issuq du motif trouvé par la première expression.

Prenons l'application suivante : nous souhaitons, dans mon fichier passwd remplacer tous les logins, quels qu'ils soient, par « user_*login* » ainsi, l'utilisateur *root* deviendra *user_root*. Nous allons alors utiliser la commande sed de la façon suivante :

cat /etc/passwd | sed -e 's/^([a-Z]|[0-9])+(:.*)$/user_\1\2/'

L'expression de recherche est la suivante: ^([a-Z]|[0-9])+(:.*)$

Elle décrit que chaque ligne (^ ... $) est constituée de deux blocs : le premier est le login, comme nous l'avons vu précédemment, le second est une chaîne quelconque, débutant par « : » et suivant le login.

L'expression de remplacement est la suivante : user_\1\2

Il s'agit tout simplement de remplacer chaque ligne du fichier *passwd* par la chaîne « user_ » suivi du rappel du premier bloc (à savoir le login), suivi du rappel du second bloc.

Ainsi, les lignes seront bien conservées à l'identique, au point près que la chaîne « user_ » sera ajoutée au login - il est vrai que nous aurions pu écrire la même chose plus simplement, mais c'est pour l'exemple.

Un autre exemple de remplacement peut concerner la date : la fonction **date** du shell nous retourne la date sous la forme suivante : ven sep 26 11:40:15 CEST 2003 que nous pourrions souhaiter afficher de la sorte : Nous sommes le ven. 26 sep. 2003. L'usage de la commande sed rendra l'opération simple par la commande suivante (moins simple elle) :

date | sed -e 's/^\([^]*\) \([^]*\) \([^]*\).* \([^]*\)$/Nous sommes le \1. \3 \2. \4/'

Nous avons donc décrit la recherche de la façon suivante: ^([^]*\) \([^]*\) \([^]*\).* \([^]*\)$

Il s'agit en fait de définir quatre blocs sur la ligne, chacun correspondant à un mot, c'est à dire une suite de lettres différentes de l'espace, séparées par des espaces. Sur l'ensemble de la ligne, nous retenons les trois premiers mots ainsi que le dernier : l'heure et le méridien sont ainsi identifiés par le motif « .* » signifiant qu'il peut y avoir n'importe quoi à cette position.

Dans la chaîne de remplacement, nous venons simplement repositionner les quatre blocs précédemment identifiés dans un ordre différent en ajoutant les « . » et quelques mots supplémentaires.

Il est à noter que dans les exemples nous avons réutilisé tous les blocs, mais qu'il n'y a aucune obligation à cela.

Sed peut s'utiliser avec une option « globale » indiquant que s'il y a plusieurs substitutions sur la même ligne elles pourront alors être faites – sans quoi sed s'arrête après la première. Cette option s'ajoute de la façon suivante : sed -e 's/expression_recherchée/expression_de_remplacement/**g'.**

Sous vi, il est possible d'utiliser la même syntaxe pour effectuer une opération de rechercher-remplacer utilisant des expressions régulières : en mode commande, tapez

:% s/expression_recherchée/expression_de_remplacement/

Il est possible de préciser sur quelles lignes :

:ligne_debut,ligne_fin s/expression_recherchée/expression_de_remplacement/

Les Scripts shell

Les scripts shell sont la réponse à l'automatisation : nous avons vu au travers de l'usage des commandes du shell les possibilités que nous avons pour lancer des programmes, manipuler des fichiers, des données. Nous avons vu comment enchaîner ces commandes mais nous n'avons pas vu comment sauvegarder un enchaînement pour le rejouer à volonté par la suite. La réponse à cette

question est l'écriture de scripts shell.

Un script shell est simplement un fichier texte contenant la liste des commandes à exécuter, on trouvera sur chaque ligne de nouvelles commandes, telles que nous les taperions dans un terminal. Outre le simple enchaînement de commandes, nous allons profiter de ce chapitre pour découvrir les schémas classiques de programmation, dans le cadre du shell, à savoir les alternatives et les itérations.

Contrairement aux programmes en C, les scripts shell ne sont pas des programmes compilés. Il s'agit de programmes interprétés, c'est à dire que les lignes sont évaluées les unes après les autres : il n'y a pas de vérification syntaxique générale avant l'exécution par exemple. Ce n'est donc que lors de l'exécution que vous saurez si la syntaxe des parties en cours d'évaluation est bonne ou non.

La création de scripts shell se fait à l'aide d'un éditeur de texte comme vi, emacs... Il n'y a rien de spécifique à ce type de programmation.

La première ligne d'un script indiquera quel shell doit être utilisé pour exécuter correctement le script : nous avons vu qu'il existait plusieurs shell différents (sh, bash, ksh...) dont la programmation varie quelque peu. La première ligne définira donc précisément comment exécuter le programme.

Cette ligne sera généralement la suivante (sous linux) : #!/bin/bash

Plus généralement on utilisera (plus standard) : #!/bin/sh

Le caractère « # » indique que la ligne contient des commentaires et ce qu'il y a derrière sera ignoré. Il n'y a donc que sur la première ligne que l'interprétation sera différente.

Voici un exemple complet pour notre premier script : « Hello World »

```
#!/bin/bash
# exemple de script shell
echo "Hello World !"
```

Ces lignes sont à enregistrer dans un fichier *helloWorld.sh* par exemple. Pour exécuter le script, il y aura ensuite deux façons de procéder :

- Appeler le shell en précisant le nom de ce script : *bash helloWorld.sh*
- Rendre ce script exécutable puis l'exécuter : *chmod +x helloWorld.sh* puis *./helloWorld.sh*

Une fois les droits d'exécution donnés au script, il peut être appelé comme n'importe quel programme du système ou comme tous les programmes que vous réalisez en C par exemple.

Le passage de paramètres

Lorsque l'on souhaite automatiser un traitement, on cherche généralement à fournir à ce traitement des paramètres. Par exemple plutôt que d'écrire deux scripts pour arrêter et relancer une application (*start.sh* et *stop.sh)* nous allons généralement préférer écrire un seul script prenant en paramètre l'action à exécuter. Ainsi le script s'appellera *monApplication.sh* et prendra en paramètre soit *start* soit *stop*. Pour démarrer l'application, nous taperons donc la commande : *./monApplication.sh start*

La chaîne *start* est donc la valeur que prendra le premier paramètre du script. Un script pourra prendre autant de paramètres que nécessaire. Seules des contraintes de taille maximale de ligne, applicables différemment selon les systèmes UNIX, viendront limiter ce nombre.

Au lancement du script, les paramètres de la ligne de commande sont automatiquement

affectés à des variables : « **$1** », « **$2** », « **$3** » ... « **$9** » pour respectivement les premiers, seconds, troisièmes... et neuvièmes paramètres. Nous verrons par la suite comment gérer les commandes à plus de neuf paramètres.

La variable « **$0** » contiendra le nom du programme lui-même. La variable « **$#** » indiquera le nombre d'arguments passés à la commande. Enfin, la variable « $* » contient l'ensemble des arguments. On peut ainsi dire que la ligne de commande saisie par l'utilisateur lors de l'appel de la commande peut être obtenue par la commande suivante : *echo $0 $**.

Lorsque la commande reçoit plus de neuf arguments, il faudra utiliser la commande « shift » pour accéder à tous les arguments. Cette fonction décale les arguments vers la gauche. Ainsi, $1 prendra pour valeur celle qu'avait $2 ; $2 prendra anciennement $3 et ainsi de suite. La valeur qui était dans $1, avant l'appel de *shift* est donc perdue.

Voyons un petit exemple d'illustration:

```
#!/bin/bash
# Un programme d'illustration pour le passage de paramètres
echo "Le programme appelé est $0"
echo "Le nombre d'arguments passé est $#"
echo "Les 9 premiers arguments sont : $1 $2 $3 $4 $5 $6 $7 $8 $9"
shift
echo "Le 10ème argument est : $9"
```

Les structures algorithmiques classiques

Il n'y a que deux structures algorithmiques nécessaires à la réalisation de tout programme, la première est l'alternative, la seconde l'itérative. Ces structures s'écrivent de multiples façons selon le shell que l'on utilise, nous verrons dans ce chapitre la forme la plus classique et la plus courante, celle du shell de base : ***sh***.

L'alternative s'écrit de la façon suivante :

```
if expression
then
        instructions
else
        instructions
fi
```

Il n'y a pas, en shell, d'accolades, ce sont les mots clefs **then**, **else** et **fi** qui indiqueront le début et fin de bloc.

Il existe une variante permettant de simplifier l'écriture des tests imbriqués utilisant le mot clef **elif**.

```
if expression
then
        instructions
elif expression
then
```

instructions

else

instructions

fi

L'itérative classique s'écrit sous la forme suivante :

while *expression*

do

instructions

done

Nous verrons par la suite qu'il existe une autre forme de boucle très utile et très spécifique à l'environnement shell.

Les expressions de test

Dans ces structures, nous avons vu apparaître la notion *d'expression*, il s'agit du test qui va déterminer si l'on doit exécuter la partie **then** ou **else.** Concrètement, cette expression correspond à l'appel d'une fonction ou d'un programme shell. Comme pour tout programme, si le code de retour de cet appel est « 0 » alors le test sera positif et la partie **then** sera exécutée. Si le code de retour est différent de « 0 » c'est alors par le **else** que le programme continuera.

Dans l'exemple suivant, le message « l'utilisateur root existe » sera affiché lorsque celui-ci sera déclaré dans le fichier */etc/passwd.*

```
#!/bin/bash
if egrep "^root:" /etc/passwd >/dev/null
then
        echo "L'utilisateur root existe"
fi
```

Lorsque egrep trouve une ligne correspondant au motif indiqué, la commande retourne « 0 » ; le programme continue alors son exécution par le bloc « **then** » et le message est affiché.

Vous ferez bien attention de ne pas mélanger les notions de « code de retour » avec ce que l'application affiche sur sa sortie standard. Le « code de retour » est bien le code erreur éventuel retourné par le programme, celui qui est obtenu en affichant la variable « ? » (*echo $?*).

La majeure partie des tests que vous allez effectuer sont des tests de comparaisons de valeurs numériques ou de chaînes de caractères. Il existe pour cela une commande appelée *test*. Voyons par exemple comment vérifier que le nombre de paramètres reçus par le programme est bien égal à 5.

```
#!/bin/bash
if test $# -ne 5
then
        echo "Le nombre d'arguments doit être égal à 5"
fi
```

La commande *test* reçoit en arguments ce que l'on souhaite vérifier : ici que **$#** n'est pas égal (**ne**) à **5**. Cette commande retourne **0** lorsque le test est vrai **1** sinon. La commande *test* pourra être appelée différemment en utilisant des « [ ] » Cette façon sera, en réalité, la plus couramment utilisée. Ainsi l'exemple précédent pourra être écrit :

```
#!/bin/bash
if [ $# -ne 5 ] ; then echo "Le nombre d'arguments doit être égal à 5" ; fi
```

Rq : il sera important de laisser des espaces avant et après les caractères « [» et «] ».

La fonction test permet de réaliser tous les tests numériques classiques ainsi que les comparaisons de chaînes :

[*nb1* **-eq** *nb2*]	- Retourne vrai si le nombre *nb1* est **égal** au nombre *nb2.*
[*nb1* **-ne** *nb2*]	- Retourne vrai si le nombre *nb1* est **différent** du nombre *nb2.*
[*nb1* **-gt** *nb2*]	- Retourne vrai si le nombre *nb1* est **supérieur strict** au nombre *nb2.*
[*nb1* **-ge** *nb2*]	- Retourne vrai si le nombre *nb1* est **supérieur ou égal** au nombre *nb2*
[*nb1* **-lt** *nb2*]	- Retourne vrai si le nombre *nb1* est **inférieur strict** au nombre *nb2.*
[*nb1* **-le** *nb2*]	- Retourne vrai si le nombre *nb1* est **inférieur ou égal** au nombre *nb2.*
[*ch1* == *ch2*]	- Retourne vrai si les chaînes *ch1* et *ch2* sont **égales.**
[*ch1* **!=** *ch2*]	- Retourne vrai si les chaînes *ch1* et *ch2* sont **différentes.**
[**-z** *ch1*]	- Retourne vrai si la chaîne *ch1* est **vide.**
[**-n** *ch1*]	- Retourne vrai si la chaîne *ch1 n'*est pas **vide.**

Les possibilités de la commande *test* ne s'arrêtent pas là, ainsi elle permet de tester de nombreuses choses comme l'existence de fichiers :

[**-d** *fic*]	- Retourne vrai si *fic* est un répertoire.
[**-e** *fic*]	- Retourne vrai si *fic* est un fichier qui existe.
[**-f** *fic*]	- Retourne vrai si *fic* est un fichier simple.
[**-x** *fic*]	- Retourne vrai si *fic* existe et est exécutable.
[**-w** *fic*]	- Retourne vrai si *fic* existe et est accessible en écriture.
[**-r** *fic*]	- Retourne vrai si *fic* existe et peut être lu.

Le manuel vous permettra de connaître la liste de tous les tests possibles. Chacun des tests peut être inversé par l'insertion du caractère « **!** ». Ainsi nous pouvons obtenir :

[**!** **-e** *fic*]	- Retourne vrai si *fic* n'existe pas.

Il sera possible d'enchaîner les tests comme nous l'avions vu précédemment pour l'enchaînement de commandes (à vrai dire, les tests étant des commandes comme les autres, il s'agit donc d'une application directe de ce que nous avons précédemment vu).

Ainsi pour réaliser un second test si le premier est réussi :

if [*test1*] **&&** [*test2*] ... soit par exemple if [-r *fic*] **&&** [-w *fic*] ; then...

La partie *then* sera exécutée si le fichier existe et s'il est accessible en lecture et en écriture.

Il est aussi possible d'exécuter un second test lorsque le premier a échoué :

if [*test1*] || [*test2*] ... soit par exemple if [! -e *fic*] || [! -s *fic*] ; then...

La partie *then* sera exécutée si le fichier *fic* n'existe pas ou si ce même fichier est vide.

Voyons un exemple un peu plus complet.

Écrivons un script heure qui reçoit en paramètre des heures, minutes, secondes, vérifie que ces nombres soient dans le bon intervalle et affiche l'heure sous la forme : "il est *heure*h*minutes* et *secondes* secondes".

```
#!/bin/bash
if [ $# -ne 3 ] ; then
  echo "Le nombre d'arguments n'est pas correct : "
  echo "usage : $0 heure minutes secondes" ; exit 1
else
   h=$1; m=$2; s=$3
   if [ $h -ge 0 ] && [ $h -le 23 ] && [ $m -ge 0 ] && [ $m -lt 60 ] && [ $s -ge 0 ] && [ $s -lt 60 ]
   then
      echo "il est ${h}h${m} et ${s} secondes" ; exit 0;
   else
     echo "paramètres incorrects"  >&2
  fi
fi
```

Rq : ici, nous avons utilisé une notation particulière pour l'utilisation des variables : nous les avons mises entre « { } » Le but est d'indiquer au shell où commence et finit le nom de la variable. Sans cela, nous aurions écrit $hh ce qui signifie la variable « hh » et non la variable « h » suivie du caractère « h ». L'utilisation des « { } » doit être un réflexe de sécurité que vous utiliserez aussi souvent que possible.

Plus de boucles

Nous avons vu l'itération classique de type *while*, comme dans tout langage, il existe plusieurs formes d'itérations.

L'itération classique :

while *expression ;* **do**

instructions

done

Comptons par exemple de 0 à 9 :

i=0 ; **while** *[$i -lt 10] ;* **do**

instructions

i=$(($i +1))

done

La répétition jusqu'à ce que la condition soit vraie :

until *expression* ; **do**

instructions

done

Comptons de 0 à 9 par exemple :

i=0 ; **until** *[$i -eq 10]* ; **do**

instructions

i=$(($i +1))

done

Enfin, il existe une boucle totalement différente des itérations classiques. Il s'agit de la boucle **for**. Cette structure permet de parcourir une liste dont chaque valeur est séparée par un espace. Ainsi il est possible de parcourir une liste de fichiers ou une liste d'arguments, ou tout autre :

for *variable* **in** *mots* ; **do**

instructions

done

Comptons de 0 à 9 par exemple :

for *i* **in** *0 1 2 3 4 5 6 7 8 9* ; **do**

instructions

done

Parcourons la liste des fichiers du répertoire courant (ici le joker '*' est interprété comme dans un terminal et sera donc remplacé par la liste des fichiers du répertoire courant) :

for *i* **in** * ; **do**

instructions

done

Parcourons la liste des arguments passés au script (il s'agit du comportement par défaut, c'est pourquoi aucun argument à for n'est nécessaire, on pourrait cependant utiliser $* ce qui reviendrait au même) :

for *i* ; **do**

instructions

done

Dans tous les exemples ci-dessus, la variable « i » prendra à chaque itération une nouvelle valeur issue de la liste.

Nous pouvons appliquer les structures itératives à une commande que nous avons précédemment

vue : la commande **read** peut être utilisée dans la lecture de fichiers et permettra ainsi de traiter chaque ligne du fichier, une à une. Pour cela nous allons utiliser un ensemble de structures que nous connaissons déjà. Cette forme est assez simple, mais elle mérite d'être bien apprise car son utilisation est fréquente.

(**while read ; do**

instructions à appliquer à la ligne contenue dans $REPLY

done) < *fichier*

Une autre variante est ainsi :

cat *fichier* | *(* **while read ; do** *instructions ;* **done)**

Nous remarquons que la commande **read** retourne 0 lorsqu'une ligne est lue sur l'entrée standard et retournera une valeur différente lorsque l'entrée standard est fermée. Ce qui se produit soit lorsque le fichier est fini dans le cas d'une redirection, soit lorsque l'utilisateur presse CTRL+D dans la console pour une saisie clavier.

La gestion des cas

L'instruction **case** permet en shell de gérer des cas et permettra ainsi de remplacer une série de if ... elif... elif... else... fi. Cette instruction correspond au *switch* du langage C. Sa syntaxe est comme suit :

case $*variable* **in**
'valeur1')
action1
;;
'valeur2' | 'valeur3')
action23
;;
'*')
action_défaut
;;
esac

Selon que la valeur de la variable sera 'valeur1' ou 'valeur2' ou autre, des actions différentes seront exécutées. La fin d'un choix est identifiée par la présence de deux points virgules « ;; ». Il est possible d'indiquer plusieurs choix possibles en les séparant par l'opérateur « | » signifiant ici « ou ». Si aucun des choix décrit ne correspond, alors le programme exécutera l'action par défaut identifiée pas le choix '*' . Le mot « **esac » (**case en verlan... quelle bande de d'jeunes ces informaticiens des années 60 !) vient fermer l'instruction **case.**

Création de menus à choix multiples

Sans doute par héritage issu d'une idée apparue un jour à quelqu'un sous sa douche, le shell possède aussi par défaut une instruction permettant la réalisation simple de menus. Pour ma part, je

considère la chose davantage comme une étrangeté pittoresque à ne surtout pas manquer, que comme quelque chose de vraiment utile – mais je serai ravi si vos expériences futures me démontrent le contraire.

La comme **select** permet donc de réaliser des menus et d'affecter le choix de l'utilisateur dans une variable, voyons simplement l'exemple suivant :

```
select choix in 'Afficher l'aide' 'Quitter' ; do
        case $choix in
                'Afficher l'aide') echo "usage ..." ;;
                'Quitter') exit 0 ;;
        esac
done
```

L'utilisateur verra les deux choix proposés affichés sur la sortie standard comme suit :

```
1) Afficher l'aide
2) Quitter
```

L'utilisateur sera alors invité à saisir '1' ou '2'.

Vous noterez aussi un point intéressant, cette instruction limitée par **do** ... **done**, ressemble étrangement à une boucle et cela tombe bien car il s'agit, en effet, d'une boucle. Cela signifie donc qu'une fois le traitement effectué, la question du choix est à nouveau proposée à l'utilisateur et le traitement peut recommencer.

Interprétation des arguments de la ligne commande

La plupart des programmes commencent par analyser la ligne de commande saisie par l'utilisateur, les arguments passés peuvent être des options ou des arguments nécessaires au traitement. L'analyse de la ligne permet donc au programme de déterminer son comportement aussi de vérifier que la demande faite par l'utilisateur est cohérente, c'est à dire que les arguments obligatoires sont bien présents par exemple.

Voyons l'exemple suivant qui est la commande myHead, qui appellera la commande head en passant le nombre de lignes si l'option « -n » est présente et qui affichera le nom du fichier (obligatoire) passé en argument si l'option verbose « -v » est présente. La syntaxe de cette commande sera la suivante :

```
myHead [-n ligne] [-v] fichier
```

```
#!/bin/bash
if  [ $# -lt 1 ] ; then
   echo "usage : $0 [-n ligne] [-v] fichier"
   exit 1
fi
verbose='off'
optionsList=''
```

```
while [ $# -gt 1 ] ; do
        if  [ $1 == '-v'] ; then
                verbose='on'
        elif [ $1 == '-n' ] ; then
                optionsList="$optionList -$2"
                shift
        fi
        shift
done
fichier=$1
if  [ $verbose == 'on' ] ; then
  echo "Le fichier est $fichier"
fi
head $optionsList $fichier
```

La première partie du programme va vérifier si le nombre d'arguments est correct et afficher l'aide dans le cas contraire. Vient ensuite le traitement des différents arguments. Ce que nous allons faire ici, c'est parcourir la liste des arguments en la décalant au fur et à mesure. L'idée étant qu'à chaque itération nous trouvions à la position $1 le début d'une nouvelle option. Cette façon de faire permet de ne pas avoir à se soucier de l'ordre de saisie des options par l'utilisateur. Lorsqu'une option comprend plusieurs arguments, ceux-ci pourront être consultés dans les variables $2, $3... Le traitement de ce type d'option effectuera les décalages supplémentaires, nécessaires à l'alignement sur un début d'option pour l'itération suivante en appliquant autant de **shift** nécessaires. Le passage à l'itération suivante comprend un **shift**. Ainsi, les arguments sont décalés et le nombre d'arguments ($#) est décrémenté. La boucle s'arrête quand il ne reste plus qu'un seul argument qui selon la syntaxe sera le nom du programme. La dernière partie est l'exécution a proprement parlé de la commande, elle se passe de commentaires.

Voyons une version utilisant case :

```
while [ $# -ne 0 ] ; do
        case $1 in
                '-v' | '-V')
                        verbose='on'
                        shift
                        ;;
                'n')
                        optionsList="$optionList -$2"
                        shift 2
                        ;;
                '*')
                        echo 'option non reconnue !'
```

```
                shift
                ;;
        esac
done
```

L'utilisation de l'instruction **case** a l'avantage d'être beaucoup plus lisible que l'imbrication de **if.**

Notion de séparateurs

Nous avons vu qu'il était possible avec read ou for d'utiliser des listes de mots, c'est en quelque sorte la notion de tableau du C que nous retrouvons par ces listes puisqu'en l'absence de typage des variables, il n'y a en shell, pas de tableaux.

Par défaut, une liste d'éléments est donc une chaîne de caractères dont chaque élément est séparé par un ou plusieurs espaces. Ce séparateur est en fait un caractère qui peut être redéfini à l'aide d'une variable : IFS. Ainsi, il est possible d'utiliser n'importe quel séparateur. Pour le traitement de fichiers CSV où le séparateur est le caractère « ; », ou comme dans *etc/passwd* où il est « : ». Par exemple, dans ce dernier, nous utiliserons les lignes suivantes pour traiter chaque champ de chaque ligne :

```
IFS=':' ; while read ; do
  for colonne in $REPLY do
        # traitement de l'élément $colonne
  done
done < /etc/passwd
```

Optimisation des boucles

Ce que nous allons maintenant voir est sans nul doute l'ennemi numéro un d'un enseignant d'algo mais probablement aussi le meilleur ami du programmeur expérimenté. Je veux dire par là que selon votre niveau je recommande ou non l'usage des commandes suivantes. Ce qu'il faut savoir c'est que l'usage des commandes *break* et *continue* que nous allons voir n'a jamais aucun facteur obligatoire et qu'il existe toujours une façon d'écrire un algorithme sans les utiliser. Par ailleurs, je pars aussi du principe que si ces commandes ont été ajoutées à la majeure partie des langages par des personnes avec une très grosse expérience, c'est qu'elles sont fort utiles ; et que pour ma part, ce serait manquer de respect à DM Richie que de ne pas les utiliser ; ce que je ne me permettrai pas ! Trêve de plaisanteries, si ces commandes existent, c'est pour remplacer l'usage du *goto,* que tout le monde est convaincu d'abolir, sans pour autant se résoudre à ne plus optimiser les programmes. Bref, ces commandes servent à simplifier les algorithmes et ainsi accélérer l'exécution des programmes.

La commande **break** sert à interrompre une boucle, c'est à dire que quelque soit votre position dans la boucle, l'appel de cette commande sortira le programme de la boucle, sans finir l'itération en cours et se rendra à l'instruction qui suit la boucle. Voyons par exemple son usage dans le traitement des arguments d'une commande :

```
while [ $# - gt 0 ] ; do
        if [ $1 == '-v' ] ; then
```

```
            echo 'verbose'
            shift
        elif [ $1 == '-m' ] ; then
            echo 'mode m avec valeur $2'
            shift 2
        else
            echo 'erreur de syntaxe'
            break;
        fi
    done
    echo 'apres la boucle'
```

Dans l'exemple ci-dessus, si l'utilisateur passe un argument non reconnu, à n'importe quelle position, le programme passera par le cas '*else*' le message d'erreur sera affiché et le traitement sera arrêté : de la ligne *break*, le programme continuera directement sur la ligne '*après la boucle*' sans passer par où que ce soit ailleurs.

La commande **continue** sert, elle, à interrompre l'itération en cours, sans pour autant sortir de la boucle. Lorsque le programme rencontre cette instruction, il commence donc une nouvelle itération. Imaginons par exemple que dans le traitement d'un fichier, nous souhaitions ignorer les lignes commençant par un caractère '#' et afficher les autres, nous pourrions écrire cela de la façon suivante :

```
cat fichier | while read ; do
    if echo $REPLY | grep '^#' ; then continue ; fi
    echo $REPLY
done
```

Néanmoins, la meilleure façon de réaliser la même chose s'écrira :

```
cat fichier | while read ; do if  echo $REPLY | grep -v '^#' ; then echo $REPLY ; fi ; done
```

Les fonctions

Nous avons vu comment écrire des scripts shell, comme dans tout langage, comme en C, vous pouvez toujours écrire votre programme dans un seul et même fichier, dans une seule et même fonction ; il n'y a pas de raison que ça ne fonctionne pas. Cependant, il y a de très fortes chances que ce soit illisible, redondant et impossible à maintenir à court terme. L'usage de fonction permet de découper le programme en plusieurs blocs, plusieurs fichiers pour le rendre plus simple et donc plus lisible.

Les fonctions se comportent comme des scripts shell indépendants, elles reçoivent et affectent les paramètres de la même façon et retournent comme un script, un entier indiquant si leur exécution s'est correctement déroulée ou non. Comme un script, les fonctions n'ont pas de prototype défini et les arguments ne peuvent donc pas être vérifiés. L'appel d'une fonction se fait comme un appel de programme.

Une fonction est définie comme suit :

```
mafonction() {
        commandes
        ...
        return $ret
}
```

mafonction est le nom que vous donnerez à la fonction. Les « () » indiquent donc au shell que vous définissez une fonction. Vous devrez faire attention car il n'y a jamais rien entre les « () » pour rappel, contrairement au C, on ne déclare pas le prototype de la fonction. La fonction est englobée dans des « { ... } ».

Le code de retour de la fonction est retourné à l'appelant en utilisant l'instruction **return**. La valeur retournée pourra être consultée dans la fonction appelante, comme s'il s'agissait d'un programme externe, au travers de la variable « $? ».

Il sera bien nécessaire de comparer deux types de retours différents pour une fonction, comme pour n'importe quel programme shell d'ailleurs :

- Le code retourné : il indique si la fonction s'est bien déroulée ou non. Il s'agit toujours d'un entier et il est renvoyé par l'utilisation de **return** dans une fonction ou **exit** dans un script.
- Le résultat retourné : c'est ce pour quoi la fonction a été appelée et le résultat est généralement affiché à l'écran. Ce résultat est donc transmis à l'appelant par la sortie standard, en utilisant la commande **echo**.

Voyons un exemple avec la fonction *divise* qui divise le premier argument par le second si celui-ci est différent de 0. Elle retournera comme code 1 si le diviseur est 0 ; zéro sinon. Elle retournera comme valeur le résultat de la division entière.

```
divise() {
  if [ $# -ne 2 ] || [ $2 -eq 0 ] ; then        # vérifions si le nombre d'arguments est bon
                                                # et si le diviseur n'est pas 0. Dans ce cas
        return 1                                # retourner le code 1
  else
        echo $(( $1 / $ 2))                     # retourner le résultat sur la sortie standard
        return 0                                # retourner le code 0 (succès)
  fi
}
```

L'utilisation de la fonction se fera donc ainsi :

```
retour=`divise 100 10`                          # retour reçoit le résultat émis par echo
if [ $? -eq 0 ] ; then                          # on vérifie le code de retour avant d'interpréter
        echo "le résultat est $retour"          # le résulat
else
```

```
        echo "Division par zéro"
fi
```

Une fonction doit donc vraiment être considérée comme un programme. Pensez à la commande **grep**, elle affiche sur la sortie standard le résultat : les lignes trouvées dans le fichier et elle retourne un code qui est 0 ou 1 selon que son exécution s'est ou non bien passée.

Une fonction a en visibilité toutes les variables définies dans la fonction appelante. Ceci peut semble commode, mais ce serait une grave erreur que de s'en servir car l'intérêt d'une fonction est qu'elle puisse être réutilisée dans un contexte différent. Utiliser des variables globales, comme dans un programme C rend la fonction dépendante d'un contexte donné.

Cette visibilité s'explique par le fait que les variables dans un script shell sont par défaut globales. Il en va ainsi de même pour les variables définies dans les fonctions. Ainsi, voyons l'exemple suivant :

```
exemple() {
        maVariable=2
}
maVariable=3
exemple
echo $maVariable
```

Dans cet exemple, ce qui sera affiché sur la sortie standard sera « 2 » car la variable globale *maVariable* a été pour la dernière fois affectée à cette valeur.

L'usage de variables globales rend complexe la portabilité, car si l'on utilise des variables classiques comme « i,j,k » il y a de fortes chances que l'on impacte d'autres fonctions sans nous en rendre compte. Il est donc nécessaire d'utiliser des variables locales que l'on déclare simplement avec la directive **local**. Voyons l'exemple suivant :

```
exemple() {
        local maVariable=2
}
maVariable=3
exemple
echo $maVariable
```

Dans cette nouvelle version, la valeur affichée sera « 3 » car la variable locale définie dans la fonction n'a pour portée que le temps d'exécution de la fonction qui l'a définie – en bref, à la fin de la fonction, cette variable est détruite et de par sa déclaration locale, elle sera prioritaire sur la variable globale du même nom.

Toutefois, cette variable locale se comportera comme une variable globale pour une fonction appelée, comme l'illustre l'exemple suivant:

```
exemple2() {
        echo $maVariable
}
exemple() {
        local maVariable=2
        exemple2
}
maVariable=3
exemple
```

Ici, la fonction exemple2 affichera comme valeur « 2 » car l'appel d'*exemple2* est inclus dans celui d'*exemple*, le contexte de cette dernière fonction est donc transmis à la fonction appelée qui hérite ainsi des variables locales comme si elles étaient globales.

Il n'y a donc qu'une conclusion à donner : **pour s'assurer qu'une fonction soit portable, il est impératif qu'elle déclare toutes ses variables comme locales et qu'elle prenne toutes ses valeurs depuis les paramètres qui lui sont passés ; sans jamais utiliser de variables globales.**

Le dernier point qu'il reste à voir concerne les entrées-sorties. Pour les fonctions comme pour les instructions et les commandes, les entrées et sorties sont héritées du père, ici de l'appelant ; ainsi si l'on redirige la sortie d'un script, les sorties des fonctions internes au script seront elles aussi redirigées... Nous n'en attendions pas moins.

Enfin, l'appel d'une fonction ne crée pas de processus supplémentaire, sauf dans le cas où la fonction est appelée au travers d'un tube « | » ; dans ce cas un processus dédié est créé pour permettre l'exécution parallèle que nous avons déjà décrite.

Inclusion de scripts

Une programmation propre nécessite de l'ordre et l'ordre nécessite la décomposition en fonctions, comme nous l'avons vu, mais aussi la décomposition en plusieurs fichiers qui seront inclus. Ces fichiers de fonctions pourront correspondre à des librairies qui pourront être partagées dans le cadre de multiples projets par exemple.

Imaginons une librairie définie ainsi et stockée dans le fichier « lib.sh » :

```
helloWorld() {
        echo "Hello World"
}
```

Puis le programme principal utilisant cette librairie :

```
. ./lib.sh                      # permet d'inclure le fichier lib.sh
main() {
```

```
        helloWorld
}
main
```

L'inclusion du fichier librairie est effectué lors de son appel (ce qui correspond à son exécution en fait) au sein du shell courant, ce qui est exprimé par le « . » placé devant le chemin du script « ./lib.sh » La syntaxe est donc bien : « **.** ./lib.sh ».

Pour finir

L'usage du shell est une nécessité, que ce soit aujourd'hui ou demain le programmeur devra utiliser le shell ou en tout cas, c'est en l'utilisant qu'il gagnera du temps. De part mon expérience personnelle, je constate que les développeurs de bon niveau, ceux qui aiment la programmation et attendent plus de leur travail qu'un simple salaire, se tournent irrémédiablement vers l'utilisation d'un système leur permettant une manipulation par la ligne de commande. Ils ont ainsi toute la facilité du mode graphique à leur disposition et les possibilités supplémentaires d'automatisation d'un shell. C'est ainsi que Microsoft inclut maintenant un langage de shell dans ses versions professionnelles et que Mac et Linux ont fait une percée significative chez les développeurs ; ce qui est particulièrement impressionnant lorsque l'on prend en considération que sur un clavier mac, pour ouvrir une accolade « { } » il faut utiliser des touches de contrôles peu pratiques...

Voir en l'usage du shell, au travers de l'exemple de Linux, une occasion d'opposition entre le système libre et le monde de Microsoft serait fortuit ; le shell ne s'oppose pas au monde graphique, il est un outil supplémentaire qu'un informaticien doit savoir utiliser. De nombreux programmes qui demanderaient des centaines de lignes dans un langage de haut niveau peuvent être écrits en quelques dizaines de lignes en shell ; il faut donc savoir utiliser cet outil à bon escient. Il est clair que les scripts shell ne résolvent sans doute que 5% des problèmes informatiques, mais pour ces 5% ils peuvent s'avérer très efficaces.

Enfin, shell ne signifie pas « sous programme » ou « sous langage », ce n'est pas parce qu'ils sortent directement de l'ère du mode texte qu'ils ne méritent pas une programmation propre, nous avons vu par l'usage de fonctions et l'inclusion de librairies qu'il est possible de réaliser des scripts de qualité. Alors n'oubliez pas d'indenter vos scripts et de les organiser proprement. Les scripts shell sont souvent utilisés de façon secondaire pour lancer de beaux programmes bien écrits, mais il n'empêche que tout beau programme qu'il est, si son lanceur plante, il ne s'exécutera jamais correctement. En bref, le script shell est tout aussi important que n'importe quel programme écrit dans un tout autre langage. Il faut reconnaître une faiblesse au script shell : ils sont conçus dans l'optique de faire confiance au programmeur, c'est à dire qu'ils ne vérifient rien et c'est donc au programmeur de s'assurer par exemple que les arguments d'une fonctions sont bien présents ; alors c'est à vous de voir :

- ou vous êtes un programmeur expert et vous ne vérifiez rien,
- ou vous êtes un programmeur débutant et vous avez intérêt à intégrer des garde-fous vérifiant tout ce qui peut être vérifié.

J'imagine que si vous lisez ces lignes vous faites plutôt partie de la seconde catégorie, je vous invite donc à suivre mon conseil : à passer un peu plus de temps à écrire des sécurités supplémentaires, on en gagne deux fois plus en débogage évité, tout en apprenant plus rapidement. Reste la catégorie des chanceux, ceux qui se savent débutants mais vont tenter le coup : à l'université ils ont souvent le droit à une seconde chance l'année suivante, mais en entreprise ils font malheureusement perdre beaucoup de temps aux autres.