

OPTIMISER SON UTILISATION D'UNIX

BERNARD TATIN

BERNARD.TATIN@OUTLOOK.FR

RÉSUMÉ. Ce document vient des tréfonds de l'espace temps. Il a débuté il y a bien plus de trois ans de cela, repris de manière plus systématique et se trouve fortement complété aujourd'hui.

La première partie rappelle (rapidement) l'histoire et les concepts principaux des *SHELLS*. La deuxième partie est très orientée sur la recherche de *qui a piraté ma machine* mais peut être d'une grande utilité pour les débutants. La troisième partie, quant à elle, se focalise sur les scripts. Une quatrième partie donnera des notions des outils indispensables pour utiliser correctement son système UNIX.

Ce document et ses sources en ~~TeX~~ sont disponibles sur [GitHub](#).

Première partie - a. l'histoire et les concepts	2
- a.1. Une histoire d'UNIX	2
- a.2. Les <i>SHELLS</i>	3
- a.2.1. Le fonctionnement	5
- a.2.2. Quelques <i>SHELLS</i> célèbres	6
- a.2.2.1. <i>sh</i> , le Bourne <i>SHELL</i>	6
- a.2.2.2. <i>csh</i> , le C <i>SHELL</i>	6
- a.2.2.3. <i>tcsh</i> ou le <i>csh</i> interactif	6
- a.2.2.4. <i>ksh</i> , le Korn <i>SHELL</i>	7
- a.2.2.5. <i>zsh</i> , le Z <i>SHELL</i>	7
- a.2.2.6. <i>bash</i> , Bourne Again <i>SHELL</i>	7
Deuxième partie - b. la configuration et la ligne de commande	7
- b.1. La configuration	7
- b.1.1. Le shell personnel	7
- b.1.2. Configurer le prompt	8
- b.2. La ligne de commande	11
- b.2.1. Les boucles	11
- b.2.1.1. La boucle <i>for</i>	11
- b.2.1.2. La boucle <i>while</i>	12
- b.2.2. Surprises avec <i>stat</i> , <i>find</i> et <i>xargs</i>	12
- b.2.2.1. <i>stat</i>	12

Date: Novembre 2013/Novembre 2016.

- b.2.2.2.	Réfléchissons un peu	13
- b.2.2.3.	Tous les fichiers du monde	14
- b.2.2.4.	Application pratique	15
- b.2.3.	Les surprises de sudo	17
- b.2.4.	POSIX et GNU	19
Troisième partie - c. les scripts et les exemples		20
- c.1.	Les scripts <i>SHELL</i>	20
- c.1.1.	Structure des scripts	20
- c.1.2.	Choisir son <i>SHELL</i>	21
- c.1.3.	Les paramètres des scripts	21
- c.1.4.	Tests et boucles	21
- c.1.5.	Conditions, valeurs de retour des programmes	22
- c.1.6.	Redirections et tubes (ou <i>pipes</i>)	22
- c.2.	Exemples de manipulation de texte	23
- c.2.1.	Des stats	23
- c.2.2.	Peut-on faire mieux ?	25
- c.2.2.1.	Les options	25
- c.2.2.2.	Avec bash	30
Quatrième partie - d. commandes utiles		32
- d.1.	les noms de fichiers	32
- d.2.	cherche et remplace	33
- d.2.1.	cherche	33

Table des matières

Première partie - a. l'histoire et les concepts

- a.1. UNE HISTOIRE D'UNIX

Voici une (rapide) histoire d'UNIX, choisie parmi d'autres, parmi celles qui évoluent avec le temps autant parce que des personnages hauts en couleur et ayant réussi à voler la vedette à de plus modestes collègues se font effacer eux-même par de plus brillants qu'eux, soit parce que, vieillissant ils se laissent aller à des confidences inattendues.

En nous basant sur [Brève histoire d'UNIX](#), on rappelle que *AT&T* travaillait à la fin des années 60, sur un système d'exploitation **MULTICS** qui devait révolutionner l'histoire de l'informatique. Si révolution il y eut, ce fut dans les esprits : de nombreux concepts de ce système ont influencés ses successeurs, dont UNIX. Ken Thompson et Dennis Ritchie des fameux *Bell Labs* et qui travaillaient (sans grande conviction, semble-t-il) sur **MULTICS**, décidèrent de lancer leur propre projet d'OS :

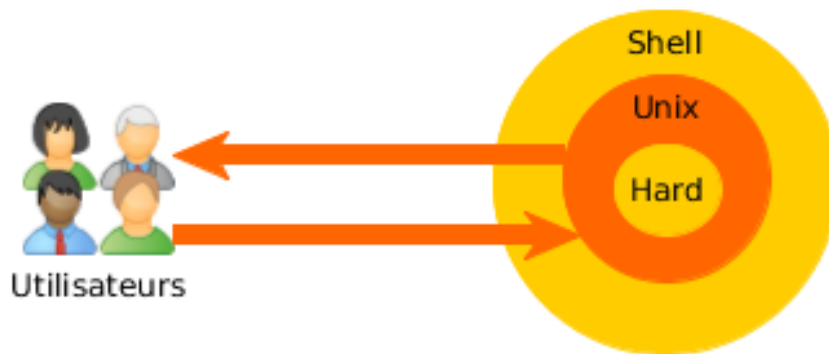
baptisé initialement *UNICS* (*UNiplexed Information and Computing Service*) jeu de mot avec "eunuchs" (eunuque) pour "un *MULTICS* emasculé", par clin d'œil au projet *MULTICS*, qu'ils jugeaient beaucoup trop compliqué. Le nom fut ensuite modifié en *UNIX*¹.

L'essor d'*UNIX* est très fortement lié à un langage de programmation, le C. À l'origine, le premier *UNIX* était écrit en assembleur, puis Ken Thompson crée un nouveau langage, le B. En 1971, Dennis Ritchie écrit à son tour un nouveau langage, fondé sur le B, le C. Dès 1973, presque tout *UNIX* est réécrit en C. Ceci fait probablement d'*UNIX* le premier système au monde écrit dans un langage portable, c'est-à-dire autre chose que de l'assembleur².

Ce que j'ai surtout retenu de tout cela, c'est qu'*UNIX* a banalisé autant l'utilisation des stations de travail connectées en réseau que le concept de *shell*, des systèmes de fichiers hiérarchisés, des périphériques considérés comme de simples fichiers, concepts repris (et certainement améliorés) à *MULTICS* comme à d'autres. Pour moi, la plus grande invention d'*UNIX*, c'est le langage C qui permet l'écriture des systèmes d'exploitations et des logiciels d'une manière très portable. N'oublions pas qu'aujourd'hui encore, C (mais pas C++) est un des langages les plus portables, même s'il commence à être concurrencé par Java par exemple.

- a.2. LES SHELLS

Un *SHELL* est une *coquille*, pour reprendre la traduction littérale, autour du système d'exploitation. Voici un magnifique diagramme (d'après ce que l'on trouve sur le WEB comme dans d'anciens ouvrages) donnant une idée du concept :



shell UNIX

Source: le WEB, ouvrages divers

Entre mes débuts dans le monde de l'informatique et aujourd'hui, le concept de *SHELL* a quelque peu évolué. Certains qualifient l'explorateur de Windows comme un *SHELL*. Ont-ils raison ? Certainement si l'on se réfère à l'image précédente : nos *commandes* (clique, clique et reclique)

1. cf l'article [MULTICS](#) de Wikipedia
2. cf [Brève histoire d'UNIX](#)

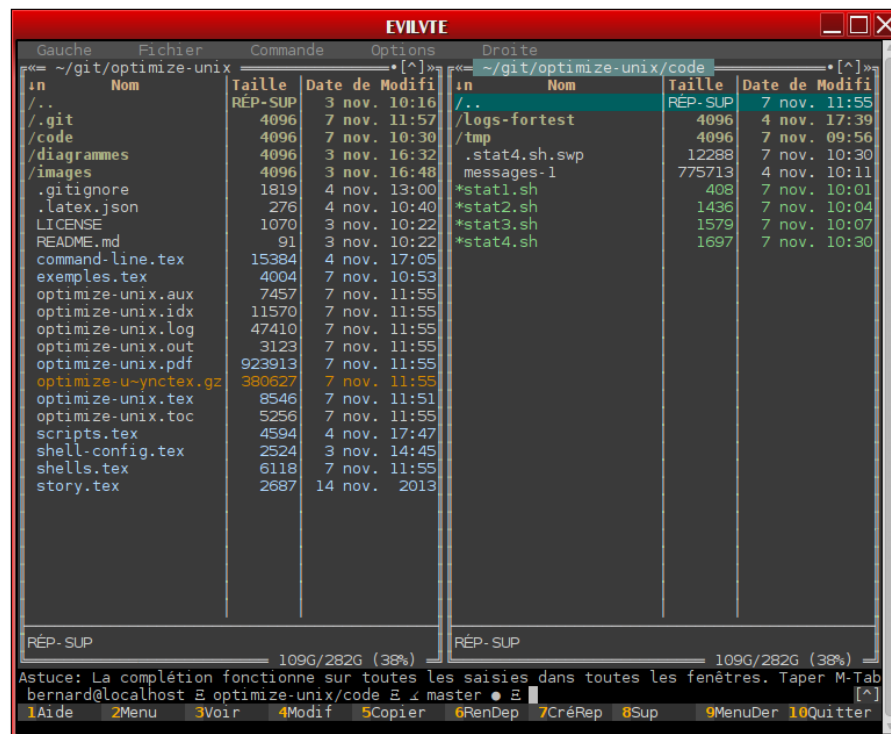
envoyée au *shell graphique* sont transmises au noyau qui nous renvoie, par l'intermédiaire du *shell graphique*, de belles images. Il faut avouer que l'explorateur Windows est le premier contact que l'utilisateur a avec sa machine. Et sur l'article [interface système](#) de Wikipedia, on trouve cette définition :

Une interface système (SHELL en anglais) est une couche logicielle qui fournit l'interface utilisateur d'un système d'exploitation. Il correspond à la couche la plus externe de ce dernier.

Ce même article cite les :

shells graphiques fournissant une interface graphique pour l'utilisateur (GUI, pour Graphical User Interface)

Dans le monde UNIX, le concept de *SHELL* reste plus modeste, même si *Midnight Commander* (mc) est parfois considéré comme un *SHELL* :



mc dans une session Cygwin

Source: mon PC

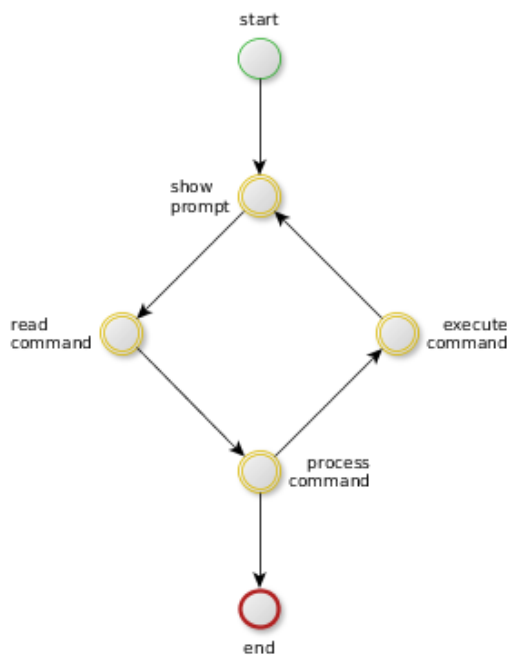
Pour nous et dans tout ce qui suit, nous considérons comme *SHELL* :

un interpréteur de commandes destiné aux systèmes d'exploitation UNIX et de type UNIX qui permet d'accéder aux fonctionnalités internes du système d'exploitation. Il se présente sous la

forme d'une interface en ligne de commande accessible depuis la console ou un terminal. L'utilisateur lance des commandes sous forme d'une entrée texte exécutée ensuite par le SHELL. Dans les différents systèmes **Windows**, le programme analogue est **command.com** ou **cmd.exe**.

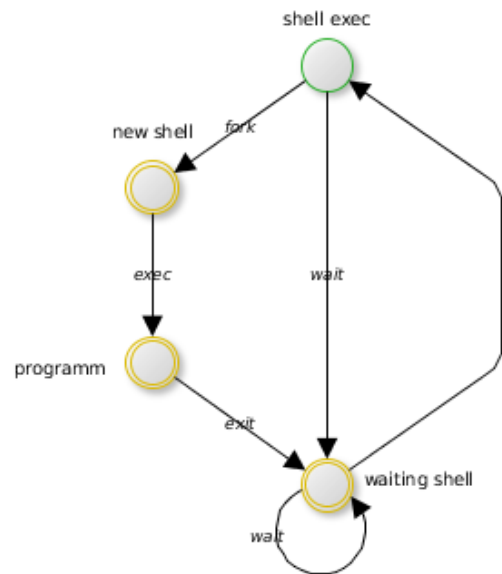
cf [SHELL UNIX](#) sur Wikipedia

- **a.2.1. Le fonctionnement.** Le fonctionnement général est assez simple, surtout si l'on ne tient pas compte de la gestion des erreurs comme dans le graphique suivant qui peut être appliqué à tout bon interpréteur. Seuls les détails de **process command** et **execute command** vont réellement changer.



SHELL : *fonctionnement général*
Source: créé avec *yEd*

L'exécution d'un programme suit l'algorithme :



SHELL : exécution d'un programme

Source: créé avec yEd

À noter que la commande **exec** se comporte différemment : elle correspond à l'appel système **exec**.

- a.2.2. Quelques SHELLS célèbres.

- a.2.2.1. *sh*, le Bourne SHELL. L'ancêtre, toujours vivant et avec lequel sont écrits une grande majorité des scripts actuels. Son intérêt essentiel est justement l'écriture de scripts. Pour l'interaction, il est absolument *nul* mais bien utile parfois pour dépanner.

- a.2.2.2. *csh*, le C SHELL. Il se voulait le remplaçant glorieux de l'ancêtre *sh* avec une syntaxe considérée plus lisible car proche du C. Il est de plus en plus abandonné y compris par ses admirateurs les plus fervents, vieillissants dans la solitude la plus complète. Essayez d'écrire un script en *csh* d'un peu d'envergure sans faire de copié/collé ! Il n'y a en effet pas de possibilité de créer des fonctions et, ce qui gêne peut-être encore plus les administrateurs système, il n'y a pas de gestion d'exception. Cependant, il fût certainement le premier à proposer l'historique des commandes.

A noter qu'il fût créé par Bill Joy, l'un des fondateurs historiques de la société Sun Microsystems.

- a.2.2.3. *tcsh* ou le *csh* interactif. Le pendant interactif du précédent. Il lui reste des aficionados qui aiment bien sa gestion de l'historique et de la ligne de commande. Il est une extension de *csh*, i.e. tout ce qui peut-être fait par *csh* est fait par *tcsh*. Sur de nombreux

systèmes (Mac OS X entre autre), ces deux *SHELLS* pointent sur le même exécutable (avec un lien symbolique).

En séquence *nostalgie*, je me souviens que c'est ce *SHELL* interactif que j'utilisais sur mon premier UNIX, en 87/88.

- a.2.2.4. *ksh*, le *Korn SHELL*. Initialement écrit pour UNIX par David Korn au début des années 80, ce *SHELL* a été repris par Microsoft pour Windows. Compatible avec *sh*, il propose de nombreuses avancées comme beaucoup de fonctionnalités de *tcsh*, des fonctions, des exceptions, des manipulations très évoluées de chaînes de caractères, ...

- a.2.2.5. *zsh*, le *Z SHELL*. C'est mon préféré pour l'interactivité, la complétion et bien d'autres choses encore dont il est capable depuis sa création ou presque. Comme *ksh*, il est compilable en bytecode et propose des bibliothèques thématiques comme la couleur, les sockets, la gestion des dates...

- a.2.2.6. *bash*, *Bourne Again SHELL*. C'est le descendant le plus direct de *sh*. C'est certainement le *SHELL* le plus répandu dans le monde Linux aujourd'hui.

Lors de ma découverte de Linux, je l'ai vite abandonné car il était très en retard pour la complétion en ligne de commande par rapport à d'autres, y compris *tcsh* qui commençait pourtant à vieillir un peu. Il a fallu beaucoup d'années (pratiquement 10) pour qu'il en vienne à peu près au niveau de *zsh*.

Aujourd'hui, c'est le *SHELL* par défaut de nombreuses distributions Linux et il commence à devenir très utilisé comme *SHELL* de script par défaut.

Deuxième partie - b. la configuration et la ligne de commande

- b.1. LA CONFIGURATION

- b.1.1. **Le shell personnel.** La première des configuration est le choix de son shell par défaut sur son compte personnel. C'est très simple :

```
1 chsh
```

Aidons-nous du manuel (sous **NetBSD**) :

```

bernard@nbsd6: man chsh
CHPASS(1)                                General Commands Manual                                CHPASS(1)

NAME
  chpass, chfn, chsh -- add or change user database information

SYNOPSIS
  chpass [-a list] [-s newshell] [-l] [user]
  chpass [-a list] [-s newshell] [-y] [user]

DESCRIPTION
  chpass allows editing of the user database information associated with
  user or, by default, the current user. The information is formatted and
  supplied to an editor for changes.

  Only the information that the user is allowed to change is displayed.

  The options are as follows:

  -a      The super-user is allowed to directly supply a user database
  entry, in the format specified by passwd(5), as an argument.
  This argument must be a colon (':') separated list of all the
  user database fields, although they may be empty.

  -s      The -s option attempts to change the user's shell to newshell.

  -l      This option causes the password to be updated only in the local
  password file. When changing only the local password,
  pwd_mkdb(8) is used to update the password databases.

  -y      This forces the YP password database entry to be changed, even if
  the user has an entry in the local database. The
  rpc.yppasswdd(8) daemon should be running on the YP master
  server.

  Possible display items are as follows:

  Login:      user's login name
  Password:   user's encrypted password
  Uid:        user's login
  Gid:        user's login group
  Change:     password change time

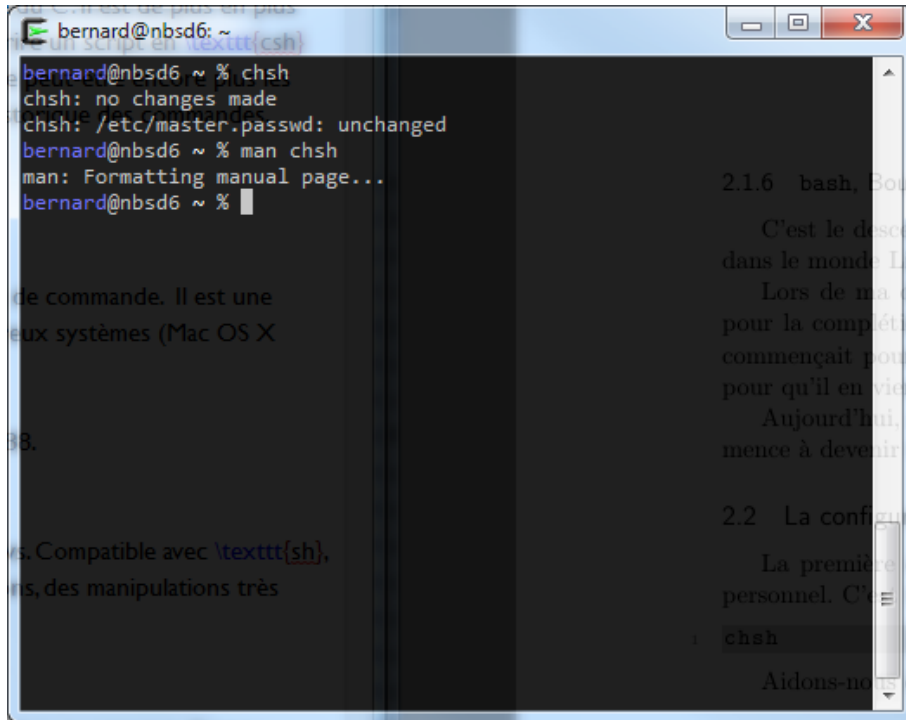
/tmp/man.01222a

```

man chsh sous *NetBSD*

Source: ma machine virtuelle

- **b.1.2. Configurer le prompt.** Sur ma machine virtuelle *NetBSD*, j'obtiens quelque chose comme ceci :

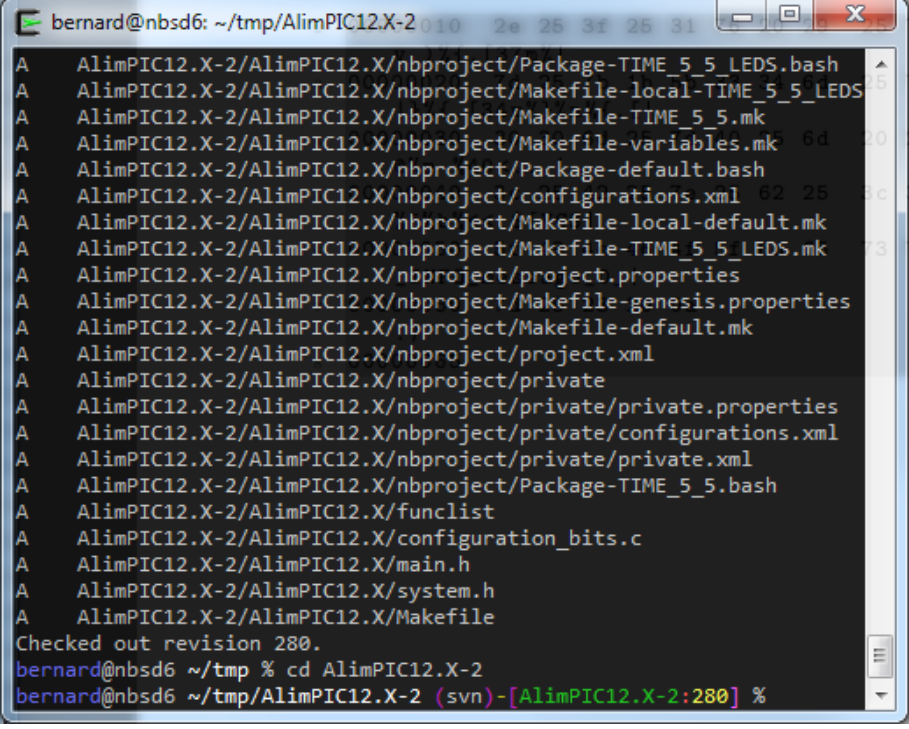


```
bernard@nbsd6: ~  
bernard@nbsd6 ~ % chsh  
chsh: no changes made  
chsh: /etc/master.passwd: unchanged  
bernard@nbsd6 ~ % man chsh  
man: Formatting manual page...  
bernard@nbsd6 ~ %
```

un prompt sous *NetBSD*, avec `zsh`

Source: ma machine virtuelle

Le prompt, ce sont les caractères colorés que l'on voit en début de chaque lignes de commande. Ce prompt m'a aidé, voire sauvé plusieurs fois. Celui-ci m'affiche le nom de l'utilisateur courant en bleu, de la machine en blanc et du répertoire courant en blanc et gras. Lorsque j'ai des sessions sur plusieurs machines, je vois tout de suite où je me trouve avec son nom. Ensuite, lorsque je me déplace de répertoires en répertoires, je n'ai pas besoin de faire d'éternels `pwd` pour savoir où je me trouve. En plus, lorsque je trouve dans un dépôt SVN, j'ai un affichage me donnant les indications sur le répertoire de travail (on ne peut pas le faire sous *CYGIN*) :



```

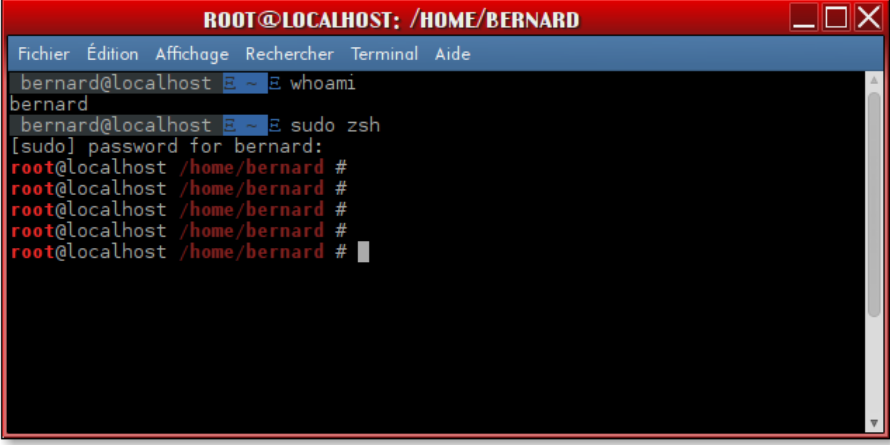
bernard@nbsd6: ~/tmp/AlimPIC12.X-2
A AlimPIC12.X-2/AlimPIC12.X/nbproject/Package-TIME_5_5_LEDS.bash
A AlimPIC12.X-2/AlimPIC12.X/nbproject/Makefile-local-TIME_5_5_LEDS
A AlimPIC12.X-2/AlimPIC12.X/nbproject/Makefile-TIME_5_5.mk
A AlimPIC12.X-2/AlimPIC12.X/nbproject/Makefile-variables.mk
A AlimPIC12.X-2/AlimPIC12.X/nbproject/Package-default.bash
A AlimPIC12.X-2/AlimPIC12.X/nbproject/configurations.xml
A AlimPIC12.X-2/AlimPIC12.X/nbproject/Makefile-local-default.mk
A AlimPIC12.X-2/AlimPIC12.X/nbproject/Makefile-TIME_5_5_LEDS.mk
A AlimPIC12.X-2/AlimPIC12.X/nbproject/project.properties
A AlimPIC12.X-2/AlimPIC12.X/nbproject/Makefile-genesis.properties
A AlimPIC12.X-2/AlimPIC12.X/nbproject/Makefile-default.mk
A AlimPIC12.X-2/AlimPIC12.X/nbproject/project.xml
A AlimPIC12.X-2/AlimPIC12.X/nbproject/private
A AlimPIC12.X-2/AlimPIC12.X/nbproject/private/private.properties
A AlimPIC12.X-2/AlimPIC12.X/nbproject/private/configurations.xml
A AlimPIC12.X-2/AlimPIC12.X/nbproject/private/private.xml
A AlimPIC12.X-2/AlimPIC12.X/nbproject/Package-TIME_5_5.bash
A AlimPIC12.X-2/AlimPIC12.X/funclist
A AlimPIC12.X-2/AlimPIC12.X/configuration_bits.c
A AlimPIC12.X-2/AlimPIC12.X/main.h
A AlimPIC12.X-2/AlimPIC12.X/system.h
A AlimPIC12.X-2/AlimPIC12.X/Makefile
Checked out revision 280.
bernard@nbsd6 ~/tmp % cd AlimPIC12.X-2
bernard@nbsd6 ~/tmp/AlimPIC12.X-2 (svn)-[AlimPIC12.X-2:280] %

```

dans un répertoire de travail **SVN**, avec **zsh**

Source: ma machine

Pour finir, le nom de l'utilisateur change de couleur lorsque je suis en **root** :



```

ROOT@LOCALHOST: /HOME/BERNARD
Fichier Édition Affichage Rechercher Terminal Aide
bernard@localhost ~$ whoami
bernard
bernard@localhost ~$ sudo zsh
[sudo] password for bernard:
root@localhost /home/bernard #
root@localhost /home/bernard #
root@localhost /home/bernard #
root@localhost /home/bernard #
root@localhost /home/bernard #
root@localhost /home/bernard #

```

en **root** avec **zsh**

Source: ma machine

Tous les shells interactifs de ma connaissance ont au moins un fichier de configuration exécuté au lancement : avec **zsh** , c'est **.zshrc** , avec **bash** , c'est **.bashrc** et avec **cs**h , c'est **.cshrc** . Aussi loin que mes souvenirs remontent, on personnalise le prompt avec la variable **PS1** et ce, même pour le MS/DOS.

Voici un hexdump de mon **PS1** :

```

1 00000000 25 7b 1b 5b 30 31 3b 33 31 6d 25 7d 25 28 3f 2e
   |%{.[01;31m}%%(?.|
2 00000010 2e 25 3f 25 31 76 20 29 25 7b 1b 5b 33 37 6d 25
   |.??%1v )%{.[37m|
3 00000020 7d 25 7b 1b 5b 33 34 6d 25 7d 25 6e 25 7b 1b 5b
   |}%{.[34m}%n%{.[|
4 00000030 30 30 6d 25 7d 40 25 6d 20 25 34 30 3c 2e 2e 2e
   |00m%}@%m %40<...|
5 00000040 3c 25 42 25 7e 25 62 25 3c 3c 20 24 7b 56 43 53
   |<%B%~%b%<< ${VCS|
6 00000050 5f 49 4e 46 4f 5f 6d 65 73 73 61 67 65 5f 30 5f
   |_INFO_message_0_|
7 00000060 7d 25 23 20 0a
                                     |}%# ./
8 00000065

```

- b.2. LA LIGNE DE COMMANDE

Pour de multiples raisons déjà plus ou moins évoquées plus haut, j'ai choisi de travailler avec *zsh* comme shell par défaut. C'est ce que nous allons faire ici, autant sous **FREEBSD** que sous **LINUX**, tout simplement pour prouver que l'utilisation du shell est assez indépendante du système sous-jacent. Mais comme je sais que certains systèmes viennent avec *bash* ou *tcsh*, sans possibilité de modification, je les évoqueraient donc, en particulier *tcsh* qui est utilisé très souvent, avec *csh*, pour l'administration. Ce n'est que fortuitement que j'examinerais *ksh*, autant par manque d'habitude que parce que je ne l'ai jamais rencontré.

- b.2.1. Les boucles. Il y a **while** et **for**.

- b.2.1.1. La boucle **for**. On commence par celle-ci car elle en a dérouté plus d'un. Nous avons, avec *zsh* et *bash*, deux syntaxes essentielles. La première *parcourt* un ensemble de données :

```

1 for file_name in *.txt
2 do
3     cat $file_name
4 done

```

Il y a la boucle plus classique pour les spécialistes de Java :

```
1 for ((i=5; i< 8; i++))
2 do
3     echo $i
4 done
```

Avec *tcsh*, nous aurons :

```
1 foreach file_name (*.txt)
2     cat $file_name
3 end
4
5 foreach i (`seq 5 1 8`)
6     echo $i
7 end
```

- *b.2.1.2. La boucle while*. Elle permet de boucles infinies comme celle-ci avec *zsh*, *bash* et *ksh* :

```
1 while true; do date ``+%T``; sleep 1; done
```

Avec *tcsh*, nous écrirons en deux lignes :

```
1 while (1); do date ``+%T``; sleep 1;
2 end
```

- *b.2.2. Surprises avec stat, find et xargs*.

- *b.2.2.1. stat*. La commande *stat* permet de connaître bon nombre de détails à propos d'un fichier comme ici :

```
1 bernard@debian7 ~ % stat *
2 install:
3 device      70
4 inode       1837064
5 mode        16877
6 nlink       3
7 uid         1000
8 gid         1000
9 rdev        7337007
10 size        512
11 atime       1383862259
12 mtime       1383085660
13 ctime       1383085660
14 blksize     16384
```

```

15 blocks    4
16 link
17
18 userstart.tar.gz:
19 device     70
20 inode      1837156
21 mode       33188
22 nlink      1
23 uid        1000
24 gid        1000
25 rdev       7371584
26 size       3498854
27 atime      1383855566
28 mtime      1383855552
29 ctime      1383855552
30 blksize    16384
31 blocks     6880
32 link

```

On obtient, sous *zsh*, un résultat totalement identique sous **NetBSD** et sous **Linux**. Si l'on fait un *which stat*, nous obtenons, sur les deux systèmes, le message *stat: shell built-in command*. C'est ce qui me plaît sous *zsh*, les commandes non standard comme *stat* sont remplacées par des fonctions dont le résultat ne réserve pas de surprise. Si je veux une sortie plus agréable et n'afficher que la date de dernière modification (cf [The zsh/stat module](#) pour de plus amples explications) :

```

1 bernard@debian7 ~ % stat -F "%Y-%m-%d %T" +mtime -n *
2 install 2013-10-29 23:27:40
3 userstart.tar.gz 2013-11-07 21:19:12
4 bernard@debian7 ~ %

1 bernard@NBSD-64bits ~ % stat -F "%Y-%m-%d %T" +mtime -n *
2 install 2013-11-14 09:52:56
3 userstart.tar.gz 2013-11-08 00:54:06
4 bernard@NBSD-64bits ~ %

```

- *b.2.2.2. Réfléchissons un peu.* Grâce à *zsh*, nous avons une méthode extrêmement portable entre Unix pour afficher des données détaillées des fichiers. Pour l'exemple, prenons le *stat* d'origine :

```

1 bernard@debian7 ~ % /usr/bin/stat --printf="%n %z\n" *
2 install 2013-10-29 23:27:40.000000000 +0100
3 userstart.tar.gz 2013-11-07 21:19:12.000000000 +0100

```

```

4 bernard@debian7 ~ %
1 bernard@NBSD-64bits ~ % /usr/bin/stat -t "%Y-%m-%d %T" -f
  "%Sc %N" *
2 2013-11-14 09:52:56 install
3 2013-11-08 00:54:06 userstart.tar.gz
4 bernard@NBSD-64bits ~ %

```

- *b.2.3. Tous les fichiers du monde.* Si je veux faire la même chose que précédemment, mais sur tous les fichiers de la machine, on peut tomber sur ce message d'erreur :

```

1 bernard@debian7 ~ % stat -F "%Y-%m-%d %T" +ctime -n $(find
  / -name "*")
2 zsh: liste d'arguments trop longue: stat
3 bernard@debian7 ~ %

```

C'est là que **xargs** entre en jeu, mais avec un nouveau problème :

```

1 bernard@debian7 ~ % find / -name "*" | xargs stat -F "%Y-%
  m-%d %T" +ctime -n
2 stat: option non valide -- F
3 Saisissez `` stat --help `` pour plus d'informations.
4 ...
5 stat: option non valide -- F
6 Saisissez `` stat --help `` pour plus d'informations.
7 123 bernard@debian7 ~ %

```

La commande **xargs** va chercher non pas la fonction de **zsh** mais le binaire qui se trouve sur le **PATH** de la machine. On doit donc faire :

```

1 bernard@debian7 ~ % find . -name "*" | xargs stat --printf
  "%n %z\n"
2 ...
3 ./w3m 2013-11-07 23:07:03.000000000 +0100
4 ./w3m/configuration 2013-11-07 23:03:48.000000000 +0100
5 ./w3m/history 2013-11-07 23:06:29.000000000 +0100
6 ./w3m/cookie 2013-11-07 23:06:29.000000000 +0100
7 ./viminfo 2013-11-07 23:07:03.000000000 +0100
8 bernard@debian7 ~ %

```

Sous **NetBSD** :

```

1 bernard@NBSD-64bits ~ % find . -name "*" | xargs stat -t "
  %Y-%m-%d %T" -f "%N %Sc"
2 ...

```

```

3 ./lessshst 2013-11-08 01:02:35
4 ./install 2013-11-14 09:52:56
5 ./zshrc.private~ 2013-11-08 01:13:05
6 ./viminfo 2013-11-08 01:14:38
7 ./xinitrc 2013-11-08 01:14:44
8 ./Xauthority 2013-11-08 01:16:25
9 bernard@NBSD-64bits ~

```

- *b.2.2.4. Application pratique.* Sur le serveur, qui est sous **LINUX**, faisons la même chose ou presque, on place la date en premier et c'est la surprise du jour :

```

1 [bigserver] (688) ~ % find /etc -name "*" | xargs stat --
   printf="%z %n\n" | sort
2 find: "/etc/ssl/private": Permission non accordée
3 stat: option invalide -- 'o'
4 Pour en savoir davantage, faites: stat --help .
5 [bigserver] (689) ~ %

```

En rajoutant l'option **-print0** à **find**, l'option **-0** à **xargs**, nous obtenons le bon résultat :

```

1 [bigserver] (689) ~ % find /etc -name "*" -print0 | xargs
   -0 stat --printf="%z %n\n" | sort
2 ...
3 2013-11-12 10:51:42.041176453 +0100 /etc/php5/conf.d/ldap.
   ini
4 2013-11-12 10:55:05.017425662 +0100 /etc/php5/cgi
5 2013-11-12 10:55:05.017425662 +0100 /etc/php5/cgi/php.ini
6 2013-11-13 14:55:28.001191244 +0100 /etc/apache2/sites-
   available/aenercom.preprod.conf
7 2013-11-13 14:56:35.601352228 +0100 /etc/apache2/sites-
   available/device.sigrenea.conf
8 2013-11-13 14:56:35.601352228 +0100 /etc/apache2/sites-
   enabled
9 2013-11-13 17:16:04.377217037 +0100 /etc/apache2/sites-
   available
10 2013-11-13 17:16:04.377217037 +0100 /etc/phpmyadmin
11 2013-11-14 01:03:38.589252417 +0100 /etc/php5/conf.d/
   mysqli.ini
12 2013-11-14 01:05:14.997350491 +0100 /etc/php5/conf.d
13 2013-11-14 01:05:14.997350491 +0100 /etc/php5/conf.d/
   mcrypt.ini

```

En fait, les noms de fichier sous UNIX peuvent contenir beaucoup de caractères étranges en dehors de /. **xargs** prend le caractère **LF** comme fin d'enregistrement de la part de son entrée standard. Si jamais un fichier contient ce caractère, plus rien ne va. Les nouvelles options permettent à **find** d'utiliser **0x00** comme séparateur d'enregistrement et à **xargs** de bien l'interpréter.

Il y a aussi une autre explication, depuis bien longtemps les outils GNU fonctionnent comme ceci et ce n'est que très récemment que le couple **find/xargs** fonctionne ainsi.

Après toutes ces considérations, on constate que le 14 Novembre 2013, un peu après 1 heure du matin, quelqu'un a modifié les fichiers **/etc/php5/conf.d/mysqli.ini** et **/etc/php5/conf.d/mcrypt.ini**, tout simplement pour remplacer les commentaires de type shell par des commentaires de type fichier ini.

Après une attaque du serveur, il est intéressant de faire le même exercice sur les répertoires vitaux comme **/bin**. Pour éviter des listings trop important, on limite la sortie à l'année 2013 et on fait une jolie boucle :

```
1 [bigserver] (694) ~ % for d in /bin /sbin /lib /lib32 /usr
   /bin /usr/sbin /usr/lib /usr/lib32; do
2 find $d -name "*" -print0 | xargs -0 stat --printf="%z %
   n\n" | egrep "^2013"
3 done | sort
```

Nous obtenons un listing fort long, correspondant aux mises à jour faites le 8 et le 12 Novembre. Maintenant que nous savons que l'attaque a eu lieu avant le 8 Novembre, on sélectionne plus sévèrement :

```
1 [bigserver] (695) ~ % for d in /bin /sbin /lib /lib32 /usr
   /bin /usr/sbin /usr/lib /usr/lib32; do
2 find $d -name "*" -print0 | xargs -0 stat --printf="%z %
   n\n" | egrep "^2013-11-0[1-7]"
3 done | sort
4 [bigserver] (696) ~ %
```

Cependant, rien ne prouve que nous n'avons pas eu de désordres un peu avant ou un peu pendant. Comme le gros des fichiers est dans **/usr/lib**, éliminons le de la liste :

```
1 [bigserver] (695) ~ % for d in /bin /sbin /lib /lib32 /usr
   /bin /usr/sbin; do
2 find $d -name "*" -print0 | xargs -0 stat --printf="%z %
   n\n" | egrep "^2013"
3 done | sort
4 ...
5 [bigserver] (696) ~ %
```


Nous n'avons des modifications qu'entre le 8 et le 12 Novembre.

Plus fort encore, afficher les fichiers modifiés ce jour :

```
1 find / -name "*" -print0 | xargs -0 stat --printf="%z %n\n" | egrep "^$(date `+%Y-%m-%d`)" | sort
```

On est débordé par l'affichage des fichiers système de Linux. Pour palier à cet inconvénient, on demande à **find** d'abandonner les répertoire **/sys** et **/proc** :

```
1 find / \( -path /proc -o -path /sys \) -prune -o -name "*" -print0 | xargs -0 stat --printf="%z %n\n" | egrep "^$(date `+%Y-%m-%d`)" | sort
```

- **b.2.3. Les surprises de **sudo**** . Reprenons l'exemple précédent en redirigeant la sortie standard vers **/dev/null** :

```
1 find / \( -path /proc -o -path /sys \) -prune -o -name "*" -print0 | xargs -0 stat --printf="%z %n\n" | egrep "^$(date `+%Y-%m-%d`)" > /dev/null
```

On aura une sortie comme celle-ci :

```
1 find: "/var/lib/postgresql/9.1/main": Permission non
   accordee
2 find: "/var/lib/sudo": Permission non accordee
3 find: "/var/cache/ldconfig": Permission non accordee
4 find: "/var/log/exim4": Permission non accordee
5 find: "/var/log/apache2": Permission non accordee
6 ...
```

Pour éliminer les **find: ... Permission non accordee**, on utilise **sudo** :

```
1 sudo find / \( -path /proc -o -path /sys \) -prune -o -name "*" -print0 | xargs -0 stat --printf="%z %n\n" | egrep "^$(date `+%Y-%m-%d`)" > /dev/null
```

C'est pire :

```
1 ...
2 stat: impossible d'évaluer ' /root/.aptitude ': Permission
   non accordee
3 stat: impossible d'évaluer ' /root/.aptitude/cache ':
   Permission non accordee
4 stat: impossible d'évaluer ' /root/.aptitude/config ':
   Permission non accordee
```

```

5 stat: impossible d'évaluer ' /root/.viminfo ': Permission
  non accordée
6 stat: impossible d'évaluer ' /root/.bash_history ':
  Permission non accordée
7 ...

```

Nous avons demandé à **sudo** de traiter **find** et avec le *pipe*, nous demandons à **xargs** de traiter les lignes de sorties avec **stat**. Ce dernier récupère un nom de fichier et le traite comme tel mais comme il n'est pas lancé avec **sudo**, nous avons ces erreurs. Essayons ceci :

```

1 sudo find / \( -path /proc -o -path /sys \) -prune -o -
  name "*" -print0 | xargs -0 sudo stat --printf="%z %n
  \n" | egrep "^$(date +%Y-%m-%d)" > /dev/null

```

C'est pas mieux, autant sous **LINUX** que sous **NetBSD** :

```

1 sudo: unable to execute /usr/bin/stat: Argument list too
  long
2 sudo: unable to execute /usr/bin/stat: Argument list too
  long
3 sudo: unable to execute /usr/bin/stat: Argument list too
  long
4 sudo: unable to execute /usr/bin/stat: Argument list too
  long
5 sudo: unable to execute /usr/bin/stat: Argument list too
  long
6 sudo: unable to execute /usr/bin/stat: Argument list too
  long

```

Il faut bien l'avouer, je ne sais pas quoi dire de plus ici - sinon noter un *TODO* : *comprendre ce qui ce passe*. Ce qui est bien avec **UNIX**, c'est qu'il y a toujours un moyen de s'en sortir. On remarquera quelques différences entre les mondes **LINUX** et **BSD**, en particulier avec **man sh**, où le premier nous renvoie sur *bash* alors que le second traite bien directement de *sh*. Dans tous les cas, **sudo sh -c "..."** est notre amie et nous obtenons avec **NetBSD** :

```

1 bernard@NBSD-64bits ~ % sudo sh -c "find / \( -path /proc
  -o -path /sys \) -prune -o -name '*' -type f | xargs
  stat -t '%Y-%m-%d %T' -f '%Sc %N' | egrep '^$(date +%Y
  -%m-%d)'" | sort "
2 2013-11-15 08:55:19 /var/run/dmesg.boot
3 2013-11-15 08:55:22 /var/log/messages
4 2013-11-15 08:55:22 /var/run/ntpd.pid

```

```

5 2013-11-15 08:55:22 /var/run/powerd.pid
6 2013-11-15 08:55:22 /var/run/sshd.pid

```

Et sous **LINUX** :

```

1 bernard@debian7 ~ % sudo sh -c "find / \( -path /proc -o -
   path /sys \) -prune -o -name '*' -print0 | xargs -0
   stat --printf='%z %n\n' | egrep '^$(date "+%Y-%m-%d")'"
2 2013-11-15 09:54:42.000000000 +0100 /var/lib/misc/statd.
   status
3 2013-11-15 09:54:41.000000000 +0100 /var/lib/urandom/
   random-seed
4 2013-11-15 09:55:09.000000000 +0100 /var/lib/dhcp/dhclient
   .em0.leases
5 2013-11-15 09:54:49.000000000 +0100 /var/lib/exim4
6 2013-11-15 09:54:49.000000000 +0100 /var/lib/exim4/config.
   autogenerated
7 2013-11-15 09:54:45.000000000 +0100 /var/lib/postgresql
   /9.1/main
8 2013-11-15 09:54:46.000000000 +0100 /var/lib/postgresql
   /9.1/main/global
9 ...

```

- **b.2.4. POSIX et GNU.** Profitons d'un moment de calme pour remarquer que de nombreuses commandes se comportent de manière très standard entre différents systèmes, y compris parfois, sous **MS/DOS**. Tout cela vient de **POSIX** ou de **GNU**.

Les guerres de religions qui opposent parfois violemment les mondes **BSD** et **LINUX**, les supporters de **Vi** et **Emacs**, ...finissent par être absorbées avec le temps et seuls quelques irréductibles les raniment, souvent plus pour s'exposer aux yeux (blasés maintenant) du petit monde concerné. Seule reste l'opposition farouche entre tenants du libre et leurs opposants.

Ici, nous avons utilisé **find** de la même manière sous **LINUX** et sous **NetBSD** ce qui n'a pas été toujours le cas, de même, **sh** se comporte de manière identique à quelques octets près sur les deux systèmes, ce qui n'était pas forcément vrai il y a quelques années. Pour revenir à **find**, nous avons un paquet de compatibilité **GNU** disponible sur plusieurs **BSD** qui reprenait le **find** que nous connaissons maintenant et l'on pouvait différencier **gfind** de **bsdfind** ³.

3. A vérifier dans les détails.

Troisième partie - c. les scripts et les exemples

- c.1. LES SCRIPTS SHELL

La magie des *SHELL* est infinie, ils nous permettent en effet de créer des programmes complets, complexes... parfois aux limites du lisible. On les appelle *scripts* pour les opposer aux applications généralement créées à partir de langages compilés mais cela ne devrait rien changer au fait qu'ils doivent être conçus avec un soin égal à celui apporté aux autres langages comme C/C++, Java...

Dans tout ce qui suit, il ne faut pas perdre de vue que le *SHELL* est une *coquille* entourant le noyau d'UNIX. Certains aspects des *SHELLS* ne font que recouvrir des appels systèmes.

- c.1.1. **Structure des scripts.** Ce qui est décrit ici est valable autant pour des langages interprétés comme l'horrible *Perl*⁴, le sublime *Scheme*⁵, le célèbre *Python* que pour n'importe quel *shell*.

La première ligne d'un script est le *shebang*. Cette ligne est très importante car elle indique de manière sûre quel interpréteur il doit utiliser pour exécuter le corps du script. Voici quelques exemples :

sh: : *#!/bin/sh*

bash: : *#!/bin/sh*

Perl: : *#!/usr/bin/env perl*

Python 2.7: : *#!/usr/bin/env python2.7*

Python: : *#!/usr/bin/env python*

awk: : *#!/bin/awk -f*

Les deux caractères **#!** sont considérés comme un nombre magique par le système d'exploitation qui comprend immédiatement qu'il doit utiliser le script dont le nom et les arguments suivent les deux caractères.

Dans un *SHELL* interactif, l'exécution d'un script suit l'algorithme suivant :

```

1 fork ();
2 if (child) {
3     open(script);
4     switch(magic_number) {
5         case 0x7f'ELF':
6             exec_binaire();

```

4. je ne suis pas objectif, mais quand même...

5. là, je me sens plus objectif... ou presque.

```

7         break;
8     case '\\#!':
9         load_shell(first_line);
10        exec_binaire(shellname, args);
11        break;
12        ...
13    }
14 } else {
15     wait(child);
16 }

```

- c.1.2. **Choisir son *SHELL*.** Par tradition autant que par prudence, on écrit ses scripts *SHELL* pour le *SHELL* d'origine, soit *sh*. Par prudence car on est certain qu'il sera présent sur la machine même si elle démarre en mode dégradé. Cependant, beaucoup de scripts sont *applicatifs* et ne pourront pas fonctionner en mode dégradé. Autant se servir d'un *SHELL* plus complet comme *bash*.

- c.1.3. **Les paramètres des scripts.** Les paramètres, leur nombre et leur taille n'ont de limites que de l'ordre de la dizaine de Ko. Il faut donc pouvoir y accéder. Le paramètre *\$0* est le nom du script parfois avec le nom du répertoire. Les neufs suivants sont nommés *\$1*, ..., *\$9*. Pour accéder aux autres il faut ruser un peu avec l'instruction *shift*.

- c.1.4. **Tests et boucles.** Les tests se font avec *if* de cette manière :

```

1 if condition
2 then
3     ...
4 else
5     ...
6 fi

```

Dans le même ordre d'idée, nous avons le *while* :

```

1 while condition
2 do
3 done

```

La construction des *condition* est tout un art, d'autant plus qu'en lieu et place du *if* nous pouvons écrire :

```

1 condition && condition_true && ...

```

ou bien :

```

1 condition || condition_false

```

Nous avons aussi une boucle **for** :

```
1 for index in ensemble
2 do
3     ...
4 done
```

La détermination de **ensemble** est assez naturelle comme par exemple avec **\$(ls *.java)**. Mais il faut être prudent : selon les *SHELLS* les résultats peuvent différer.

- c.1.5. **Conditions, valeurs de retour des programmes.** Tout les programmes sous **UNIX** s'achèvent par un **return EXIT_CODE** ou bien un **exit(EXIT_CODE)** bien senti. La valeur **EXIT_CODE** est renvoyée au programme appelant, notre *SHELL*. On peut le récupérer depuis la variable **\$\$?** puis étudier le cas :

```
1 myprogram arg1 arg2 ...
2 case \$\#? in
3     0)
4         its-okayyy
5         ;;
6     1)
7         bad_parameterzzz
8         ;;
9     2|3|4)
10        cant-open-filezzz
11        ;;
12    *)
13        unknow-error
14        ;;
15 esac
```

UNIX considère que la valeur de retour **0** est signe que tout va bien et que tout autre valeur exprime une condition d'erreur. On peut donc utiliser cette propriété ainsi :

```
1 myprogram arg1 arg2 ... || onerror ``Error code \$\#?``
```

- c.1.6. **Redirections et tubes (ou pipes).** Le premier piège dans l'utilisation des tubes dans un scripts est simple : pour chaque tube, on crée un nouveau processus. Ainsi le script suivant ne renvoie pas le résultat escompté :

```
1 #!/bin/sh
2
3 compteur=0
4
```

```

5 ls -l /bin | while read line; do
6     compteur=$(( compteur + 1 ))
7 done
8
9 printf "Il y a %d fichiers dans /bin\n" ${compteur}

1 $ ./bad1.sh
2 Il y a 0 fichiers dans /bin

```

La variable `compteur` fait partie de l'environnement de `bad1.sh`. Lorsque la boucle `while` se lance, elle est dans un nouveau contexte et sa modification se perd à la fin de la boucle. On corrige de cette manière :

```

1 #!/bin/sh
2
3 compteur=0
4
5 ls -l /bin > /tmp/ls-l.tmp
6 while read line; do
7     compteur=$(( compteur + 1 ))
8 done < /tmp/ls-l.tmp
9
10 printf "Il y a %d fichiers dans /bin\n" ${compteur}

1 ./not-so-bad1.sh
2 Il y a 173 fichiers dans /bin

```

- C.2. EXEMPLES DE MANIPULATION DE TEXTE

- C.2.1. **Des stats.** Voici un extrait d'un fichier `/var/log/messages` :

```

1 Nov  3 10:16:19 localhost org.gnome.zeitgeist.
   SimpleIndexer[2637]: ** \ldots
2 Nov  3 10:16:34 localhost org.freedesktop.FileManager1
   [2637]: Initializing \ldots
3 Nov  3 10:16:34 localhost nautilus: [N-A] Nautilus-Actions
   Menu Extender 3.2\ldots
4 Nov  3 10:16:34 localhost org.freedesktop.FileManager1
   [2637]: Initializing naut\ldots
5 Nov  3 10:16:34 localhost nautilus: [N-A] Nautilus-Actions
   Tracker 3.2.3 initializing\ldots

```

Nous voulons déterminer les moments les plus actifs de ce fichier avec une granularité de une heure. La manipulation est simple :

afficher le fichier: **cat file-name** ,

découper le fichier: **cut -d ':' -f 1** ,

trier le fichier: **sort** ,

compter les occurrences: **uniq -c** ,

trier en décroissant: **sort -n** .

Ce qui nous donne la commande :

```

1 cat $file-name |
2   cut -d ':' -f 1 |
3   sort |
4     uniq -c |
5     sort -n

```

On obtient rapidement un script (**stat1.sh**) à partir de cette ligne de commande :

```

1 #!/bin/sh
2
3 scriptname="$(basename $0)"
4
5 dohelp() {
6     cat << DOHELP
7     ${scriptname} [-h|--help] : this text
8     ${scriptname} file file\ldots : stats
9     DOHELP
10    exit 0
11 }
12
13 [ $# -eq 0 ] && dohelp
14 case $1 in
15     -h | --help)
16         dohelp
17         ;;
18     *)
19         cat "$@" | \
20             cut -d ':' -f 1 | \
21             sort | \
22             uniq -c | \
23             sort -n

```



```

24         ;;
25     esac

```

On peut tester :

```

1  $ ./stat1.sh messages -1 /var/log/messages /var/log/
    messages.1
2
3  ...
4  152 Nov 4 10
5  155 Oct 27 11
6  156 Oct 28 09
7  164 Oct 28 17
8  186 Oct 25 14
9  213 Oct 28 11
10 216 Nov 3 10
11 260 Oct 28 10
12 636 Oct 26 15
13 774 Oct 28 07
14 1770 Nov 3 09
15 3844 Oct 28 14
   26201 Oct 28 15

```

- c.2.2. **Peut-on faire mieux ?** Bien sûr ! On peut avoir d'autres options que la simple aide, on peut aussi gérer correctement les erreurs, les *signaux UNIX*...

- c.2.2.1. *Les options.* Depuis longtemps il existe une norme **POSIX** permettant de gérer les options de la ligne de commande. Malheureusement, il fut une époque où la norme avait beaucoup de variantes ce qui m'a poussé à faire ma propre gestion de ces paramètres.

Voici ma méthode, facile à mémoriser mais pas parfaite et un peu lourde :

l'aide: créer une fonction **dohelp** comme dans l'exemple précédent ; le nom **dohelp** permet d'éviter un clash avec une éventuelle commande **help**.

s'assurer de l'existence de paramètres: il suffit de faire le test **[\$# -eq 0]** et exécuter le code nécessaire.

vider la liste des paramètres: une boucle **while [\$# -ne 0]** fait l'affaire.

Voici un exemple plus parlant (script **stat2.sh**) :

```

1  #!/bin/sh
2
3  scriptname="$(basename $0)"
4
5  dohelp() {

```

```

6      cat << DOHELP
7  ${scriptname} [-h|--help] : this text
8  ${scriptname} [options] file file... : stats
9  options:
10     -s|--size N : number of most important hours, default
        5
11     -b|--byhour : for each hours
12 DOHELP
13     exit 0
14 }
15
16 size=5
17 byhour="cut -d ':' -f 1"
18 is_byhour=0
19 after=""
20
21 set_size() {
22     [ "$1" -lt 1 ] && onerror 3 "size must be > 1"
23     size=$1
24 }
25 set_byhour() {
26     byhour="${byhour} | tr -s ' ' | cut -d ' ' -f 3"
27     after=" | sort -k 2"
28     is_byhour=1
29     set_size 24
30 }
31 doit() {
32     end=1
33     cmd="cat @$@ | ${byhour} | sort | uniq -c | sort -nr |
        head -n ${size} ${after}"
34     case ${is_byhour} in
35         0)
36             printf "%-7.7s %-6.6s %-2.2s\n" "occurences" "
                date" "hour"
37             printf "%-17.17s\n" "
                -----"
38             ;;
39         1)
40             printf "%-7.7s %-2.2s\n" "occurences" "hour"
41             printf "%-10.10s\n" "
                -----"

```

```

42             ;;
43     esac
44     eval "${cmd}"
45 }
46
47 [ $# -eq 0 ] && dohelp
48 end=0
49
50 while [ $end -eq 0 ]
51 do
52     case $1 in
53         -h | --help)
54             dohelp
55             ;;
56         -s | --size)
57             shift
58             [ $# -eq 0 ] && onerror 2 "$1 needs a
                    parameter"
59             set_size "$1"
60             shift
61             ;;
62         -b | --byhour)
63             shift
64             set_byhour
65             ;;
66         *)
67             doit "$@"
68             ;;
69     esac
70 done

```

Et maintenant avec le **getopts** ⁶ :

```

1  #!/bin/sh
2
3  scriptname=$(basename $0)
4
5  dohelp() {
6      cat << DOHELP
7      ${scriptname} [-h] : this text
8      ${scriptname} [options] file file\ldots : stats

```

6. soyez prudents avec les (très) anciennes versions de *Red Hat*

```

9 options:
10     -s N : number of most important hours, default 5
11     -b : for each hours
12 DOHELP
13     exit 0
14 }
15 onerror() {
16     local exit_code=$1
17     shift
18     local error_msg="$@"
19
20     echo "ERROR: $error_msg" 1>&2
21     exit "$exit_code"
22 }
23
24 size=5
25 byhour="cut -d ':' -f 1"
26 is_byhour=0
27 after=""
28
29 set_size() {
30     [ "$1" -lt 1 ] && onerror 3 "size must be > 1"
31     size=$1
32 }
33 set_byhour() {
34     byhour="${byhour} | tr -s ' ' | cut -d ' ' -f 3"
35     after=" | sort -k 2"
36     is_byhour=1
37     set_size 24
38 }
39 doit() {
40     cmd="cat $@ | ${byhour} | sort | uniq -c | sort -nr |
41         head -n ${size} ${after}"
42     case ${is_byhour} in
43         0)
44             printf "%-7.7s %-6.6s %-2.2s\n" "occurences" "
45                 date" "hour"
46             printf "%-17.17s\n" "
47                 -----"
48             ;;
49         1)

```

```

47         printf "%-7.7s %-%2.2s\n" "occurences" "hour"
48         printf "%-10.10s\n" "
           -----"
49         ;;
50     esac
51     eval "${cmd}"
52 }
53
54 # [ $# -eq 0 ] && dohelp
55
56 [ $# -eq 0 ] && echo "you need arguments" && dohelp
57
58 while getopts "s:bh" opt
59 do
60     case $opt in
61         h)
62             dohelp
63             ;;
64         s)
65             set_size "$OPTARG"
66             ;;
67         b)
68             set_byhour
69             ;;
70         :)
71             onerror 2 "$OPTARG needs a parameter"
72             ;;
73         \?)
74             onerror 7 "option $OPTARG is unknown"
75             ;;
76     esac
77 done
78
79 shift $((OPTIND-1))
80 doit "$@"

```

En fait **getopts** ne sait traiter que les *options courtes* et classiques d'UNIX. Les *options longues* à la mode LINUX ne sont pas supportées. L'avantage de **getopts** est son mode de fonctionnement assez simple. Son inconvénient principal est d'être très spécifique à *bash* même si POSIX le soutient, ce qui fait qu'il n'est pas forcément disponible partout.

Pour avoir les *options longues*, il faut utiliser l'outil GNU **getopt** (sans le **s** de fin)⁷. Je reste donc sur ma méthode qui n'est finalement ni meilleure ni pire.

- c.2.2.2. Avec **bash**. On peut profiter des avantages de *bash* (boucles, tableaux, ...) comme dans ce script (qui ne fonctionne **pas** avec *sh*) :

```

1  #!/usr/bin/env bash
2
3  scriptname="$(basename $0)"
4  # set -e
5
6  dohelp() {
7      cat << DOHELP
8      ${scriptname} [-h|--help] : this text
9      ${scriptname} [options] file file... : stats
10     options:
11         -s|--size N : number of most important hours, default
12         5
13         -b|--byhour : for each hours
14     DOHELP
15     exit 0
16 }
17
18 size=5
19 byhour="cut -d ':' -f 1"
20 is_byhour=0
21 after=""
22 hours=
23
24 set_size() {
25     [ "$1" -lt 1 ] && onerror 3 "size must be > 1"
26     size=$1
27 }
28
29 set_byhour() {
30     local i
31     byhour="${byhour} | tr -s ' ' | cut -d ' ' -f 3"
32     after=" | sort -k 2"
33     is_byhour=1
34     set_size 24
35     for ((i=0; i<24; i++))
36     do

```

7. voir cette discussion sur [StackOverflow](#)

```

35         hours[i]=0
36     done
37 }
38
39 doit() {
40     end=1
41     cmd="cat @$@ | LC_ALL=C tr -cd '\t\n[:print:]' | ${
42         byhour} | sort | uniq -c | sort -nr | head -n ${
43         size} ${after}"
44     case ${is_byhour} in
45         0)
46         printf "%-7.7s %-6.6s %-2.2s\n" "occurences" "
47             date" "hour"
48         printf "%-17.17s\n" "
49             -----"
50         eval "$cmd"
51         ;;
52         1)
53         printf "%-7.7s %-2.2s\n" "occurences" "hour"
54         printf "%-10.10s\n" "
55             -----"
56         while read count index reste; do
57             local h=${index#0}
58             hours[h]=${count}
59         done <<<"$(eval ${cmd})"
60         for ((i=0; i<24; i++))
61         do
62             printf "%7d %02d\n" ${hours[i]} $i
63         done
64         ;;
65     esac
66 }
67
68 [ $# -eq 0 ] && dohelp
69 end=0
70
71 while [ $end -eq 0 ]
72 do
73     case $1 in
74         -h|--help)
75             dohelp

```

```

71         ;;
72     -s|--size)
73         shift
74         [ $# -eq 0 ] && onerror 2 "$1 needs a
           parameter"
75         set_size "$1"
76         shift
77         ;;
78     -b|--byhour)
79         shift
80         set_byhour
81         ;;
82     *)
83         doit "$@"
84         ;;
85     esac
86 done

```

Contrairement aux précédents, si une tranche horaire n'est pas représentée dans les fichiers logs passés en paramètres, elle sera tout de même affichée avec la valeur 0.

Quatrième partie - d. commandes utiles

- d.1. LES NOMS DE FICHIERS

Les commandes les plus utiles sont **dirname** et **basename**. La première renvoie le repertoire du nom fichier et la seconde renvoie simplement le nom de base comme ici :

```

1 $ which firefox
2 /usr/local/bin/firefox
3 $ basename $(which firefox)
4 firefox
5 $ dirname $(which firefox)
6 /usr/local/bin
7 $

```

On en profite pour présenter l'indispensable **which** qui donne le nom complet d'une application se trouvant dans le **PATH**.

- d.2. CHERCHE ET REMPLACE

Les outils de base sont **egrep** , **sed** et **tr** ⁸. Ces trois outils utilisent les expressions régulières. Ces expressions régulières ressemblent fortement à ce que l'on trouve dans Perl, Python, Java et les autres. La différence fondamentale à ne pas oublier : les expressions régulières des outils GNU sont rapides, efficaces, les autres...beaucoup moins ⁹.

Les expressions régulières méritent une formation complète car elles ne sont pas vraiment intuitives. De plus, les variations qui existent entre **POSIX**, **GNU**, **BSD**, les *SHELLS* qui en rajoutent parfois, sans compter que certains outils, certaines distributions n'ont pas leurs outils vraiment à jour (*cf* **RedHat**).

- d.2.1. **cherche**. Pour la recherche, nous avons **grep** et **egrep** . En fait, la plupart du temps, **egrep** équivaut à **grep -E** permettant l'utilisation des expressions régulières dites étendues ou **POSIX** dont la documentation se trouve dans *man 7 regex* sous **Debian**.

8. L'outil **awk** , est, à mon humble avis, à reléguer dans les musées.

9. L'introduction du *backtracking* peut détruire complètement les performances