République **A**lgérienne **D**émocratique et **P**opulaire

Ministère de l'**E**nseignement **S**upérieur et de la **R**echerche **S**cientifique

Université des **S**ciences et de la **T**echnologie d'**O**ran **M**ohamed **B**oudiaf

Faculté de Mathématiques et Informatique **D**épartement d'informatique

Polycopié TP Système d'exploitation Unix
Exercices et Quelques Corrigés
Sous RedHat Linux/Unix

Dr. Djamila BENHADDOUCHE

« Ceux qui sont férus de pratique sans posséder la science sont comme le pilote qui s'embarquerait sans timon, ni boussole et ne saurait jamais où il va »

(Léonard de Vinci 1452 – 1519)

« Unix est simple. Il faut juste être un génie pour comprendre sa simplicité. »

(Dennis Ritchie 1941 – 2011)

Je conseille donc ce polycopié à toutes les personnes qui souhaiteraient mettre leur deuxième pied dans le monde d'Unix/Linux.

« Tout le monde peut réussir, la seule façon de faire un excellent travail est d'aimer ce que vous faites. »

-Steve Jobs, le créateur d'Apple (1955-2011)-

AVANT PROPOS :

Ce polycopie a pour objectifs de :

- *Justifier l'emploi du système Unix;*
- *Faire découvrir le système d'exploitation Unix;*
- *Former le lecteur à sa pratique;*
- *Permettre à chacun d'être plus efficace dans son travail sous Unix en utilisant les commandes appropriées.*
- *A la fin le lecteur pourra être considéré comme un utilisateur averti.*

***Douglas McIlroy**, l'inventeur des tuyaux **Unix** (**Unix pipes** en anglais) et l'un des fondateurs de la tradition d'Unix, résume la **philosophie** comme suit :*

« Voici la philosophie d'Unix :

Écrivez des programmes qui effectuent une seule chose et qui le font bien.

Écrivez des programmes qui collaborent.

Écrivez des programmes pour gérer des flux de texte [en pratique des flux d'octets], car c'est une interface universelle. »

Ce qui est souvent résumé par : « Ne faire qu'une seule chose, et la faire bien. ».

- *Aussi ce polycopié généralement sur les scripts shell s'adresse aux utilisateurs des systèmes Unix/Linux souhaitant s'entraîner à l'écriture de scripts shell.*
- *Les fonctionnalités des cinq shells couramment utilisés (Bourne Shell, Korn Shell, Bourne again Shell, Cshell, Tcshell) sont exploitées dans une succession d'exercices de difficulté progressive.*
- *Les premiers chapitres vous permettront de manipuler des variables, de rédiger des scripts Shell en utilisant des structures de contrôle et de savoir déboguer un script. Vous vous exercerez à définir des variables de type tableau, des fonctions, à concevoir des menus ou encore à gérer les entrées et sorties d'un script.*
- *Quelques exercices en langage C, traduits du compilateur C vers l'interpréteur Shell et vice versa.*

Enfin ce polycopié est le fruit de plusieurs années dans le monde de différents UNIX (Ultrix Digital, Solaris Sun, et enfin Red Hat Entreprise Linux 5 sous Dell).

Toutes ces séries de travaux pratiques durant mes années d'enseignement ont donné des apports dans les deux sens que ce soit du mien comme ceux de mes étudiants.

Exemple : un de nos anciens étudiants l'un des meilleurs de sa promotion a eu l'opportunité d'enseigner le système Unix dans la plus prestigieuse université du monde MIT (Massachusetts Institute of Technology ***USA****)*

SOMMAIRE

I/ Quelques bonnes raisons pour se mettre à Unix. **01**

II/ Description et caractéristiques d'Unix. **03**

III/ Le démarrage. **04**

IV/ Où trouver de l'information sur Unix? **05**

V/ Tout faire avec et grâce aux fichiers. **05**

VI/ Des éditeurs de textes et le contrôle de la ligne de commande. **06**

VII/ Les principales commandes d'Unix. **09**

VIII/ Faire Connaissance avec le Shell. **15**

a. Introduction au Shell. **15**

b. Mécanismes essentiels du Shell. **17**

c. Quelques commandes Unix pour écrire des scripts. **18**

d. L'interpréteur de commandes ou Shell bash. **20**

e. Manipulation des variables. **22**

f. Les structures de contrôles : **22**

f.1 Les boucles conditionnelles while et until. **25**

f.2 Le choix Case. **26**

f.3 Le test if. **27**

g. Quelques commandes supplémentaires. **30**

g.1 set. **30**

g.2 Manipulation des expressions arithmétiques avec expr. **31**

g.3 exit. **32**

g.4 shift. **32**

g.5 time. **32**

g.6 Tableaux. **33**

g.7 eval. **34**

g.8 Fonctions. **34**

g.9 La commande awk. **34**

IX/ Les redirections: gestion des flux et des processus. 37

a. Série d'exemples. **37**
b. Caractéristiques de processus. **39**
c. Processus exécutés en avant ou arrière-plan. **41**
d. Ecriture de quelques scripts simples en Shell (bourne –shell). **43**
e. Ecriture de quelques scripts traduits en différents shell (bourne - shell, korn –shell, C-shell, T-Cshell). **45**

X/ Langage C sous Unix. 50

a. Rappel du langage C avec quelques exemples. **50**
b. Traductions de quelques exemples du compilateur langage C vers l'interpréteur Shell. **58**

XI/ Conclusion. 63

XII/ Références. 64

« Unix n'a pas été conçu pour empêcher ses utilisateurs de commettre des actes stupides, car cela les empêcherait aussi de réaliser des actes ingénieux. » – Doug Gwyn-

I/ Quelques bonnes raisons pour se mettre à Unix

a. Unix système universel:

Dans toute entreprise, tout laboratoire, toute université, les stations de travail ou de calcul sont sous Unix et le nombre de PC sous Linux est croissant. En sachant utiliser Unix, on peut travailler de manière identique et efficace sur toutes les plates-formes non Windows.

b. Unix système stable.

c. Certaines utilisations nécessitent des contraintes de production fortes telles que:

a. la disponibilité (pas de reboot, pas d'arrêt),
b. la performance en charge (nombre d'utilisateurs, de processus),
c. la pérennité (car Unix est basé sur des standards),
d. et la stabilité (pas ou peu de bogue système).

d. Interfaces d'Unix frustres (mais efficaces) ou riches :

Une des principales difficultés d'Unix reste son abord par ligne de commande, un peu démodé demandant un minimum d'investissement avant de pouvoir faire la moindre tâche. Ce type d'interface frustre reste pourtant inégalé en efficacité depuis 30 ans ! Par ailleurs, il existe depuis plus de dix ans des interfaces graphiques comparables à ceux du système Windows et maintenant les environnements graphiques sous Linux, par exemple KDE, qui n'ont plus rien à envier à celui des systèmes Microsoft.

e. Devenez votre propre ingénieur-système :

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

f. Ecriture de scripts de commandes :

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

g. Unix répond à vos besoins :

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

II/ Description et caractéristiques d'Unix

a. Système ouvert (pas de code propriétaire ; seules certaines implémentations sont propriétaires).

b. Multitâches (plusieurs programmes peuvent s'exécuter en même temps, sans blocage).

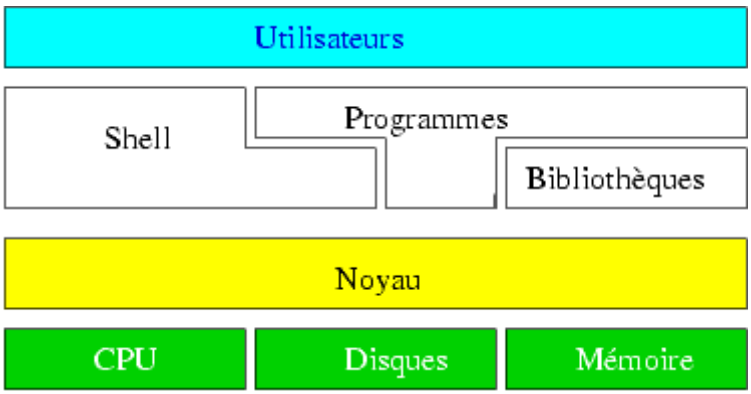
c. Mémoire protégée (pas d'interaction entre les programmes) et virtuelle (le système peut utiliser plus de mémoire que la mémoire physique disponible).

d. Multi-utilisateurs (plusieurs utilisateurs travaillent sur la même machine en même temps), gestion des droits.

e. Interactif et *batch*.

f. Interface graphique X et shell (interpréteur de commandes).

g. Plusieurs centaines d'outils (manipulation de texte, développement de logiciels, communication etc.).



Sur un système unix, on trouve deux types de personnes, celles qui vont utiliser le système (user utilisateur) et celles qui vont l'administrer (root administrateur). Les premières ont le droit d'exécuter certaines commandes propres à leur environnement et leur travail, quelques commandes liées au système leur sont

interdites. Seuls les administrateurs peuvent installer et configurer. Ils sont chargés de la bonne marche de la machine.

III/ Le démarrage.

Donc, vous voilà connecté à une machine Unix sous X (en fait X11-windows).

Sachez d'abord que vous êtes sous votre répertoire HOME. C'est un espace disque qui vous appartient, à vous et à vous seul. Normalement vous pouvez écrire et lire tous les fichiers qui s'y trouvent.

Unix est un système d'exploitation robuste. Il est impossible à un utilisateur d'arrêter involontairement un ordinateur au point que le seul remède soit un reboot. Si tout parait bloqué, on peut souvent s'en tirer avec un `Ctrl-c` *(touche* `Ctrl` *maintenue enfoncée pendant qu'on tape le* `c`*) ou avec un des boutons de la souris qui fait apparaître un menu : au pire, se déconnecter suffit à remettre tout en place*

Unix est convivial. Cependant Unix ne précise pas vraiment avec qui. »

– Steven King-

IV/ Où trouver de l'information sur Unix?

Trouver l'information n'est pas un problème, le plus dur est de se poser la ou les bonnes questions. La commande utile est *man* à faire suivre du nom de la commande inconnue. Elle permet de tout connaître sur une commande ou un produit sous Unix (comme sa syntaxe ou ses options). En utilisant l'option *-k* vous pouvez chercher un mot clé particulier plutôt qu'une commande. Si la commande n'a pas de man, essayez les options ***-?*** ou ***-h*** à l'appel de votre commande.

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

V/ Tout faire avec et grâce aux fichiers

La philosophie d'UNIX :

On peut résumer très simplement le problème par le petit dicton suivant :

Chez UNIX, tout est fichier. Ça veut dire qu'un programme, une librairie, un périphérique extérieur (disquette, disque, CDROM, imprimante) est vu par le système comme un fichier. Pour l'utilisateur débutant, les implications de ce principe sont peu importantes, mais cette unité de représentation fait d'UNIX un système très sûr et très modulable.

Un fichier peut avoir 4 contenus:

- il peut représenter tout d'abord des données ou un programme;

- il peut aussi représenter un répertoire (d) ! (c'est un fichier rempli d'index);
- il peut être également un lien symbolique (l), c'est à dire pointer sur un autre fichier (même plus haut dans l'arborescence, cycle possible);
- il peut posséder un statut particulier (device) lorsqu'il permet d'accéder à un périphérique (disque, carte son etc.).

Remarques sur le nom des fichiers:

- un nom de fichier qui commence par un / est dit nom absolu, il est nommé en partant de la racine et en descendant dans les répertoires suivants.
- un fichier qui ne commence pas par un / est dit nom relatif, il est recherché à partir du répertoire courant.
- il n'y a pas de limitation sur le nom des fichiers (à part les caractères spéciaux / ([%#$...) qu'il est préférable d'éviter!)

VI/ Des éditeurs de textes et le contrôle de la ligne de commande

vi/emacs/xemacs/nedit

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

Chacun a ses propres avantages et défauts, aucun des deux n'est vraiment simple au début.

Pour ma part, j'utilise toujours **vi** ! (enfin vim, version améliorée de vi). Je le conseille vivement à mes étudiants !

Ceci dit, ***nedit*** est également disponible sur la plupart des machines et ne pose aucun problème au débutant (sauf pour le trouver, il est peut être sous

/usr/bin/X11/nedit). Il s'utilise comme un traitement de texte et colorie les mots clés, par contre, il n'offre pas toutes les possibilités de ***emacs*** ou ***vi*.**

Les principales commandes sous ***vi*.**

Remarque : pour basculer d'un mode à un autre on appuie toujours sur la touche Esc du clavier (AZERTY OU QUERTY).

Commandes	Fonctions
I	**basculement en mode insertion**
A	**basculement en mode ajout**
Cw	**modification du mot courant**
Esc	**sortie du mode d'insertion/ajout/modification**
J	**concaténation de la ligne courante et de la suivante**
X	**effacement du caractère suivant**
X	**effacement du caractère précédent**
$	**déplacement en fin de ligne**
0	**déplacement en début de ligne**
Dd	**suppression de ligne et copie dans le buffer**
Dw	**suppression de mot et copie dans le buffer**
P	**copie du buffer sous la ligne courante**
D	**effacement jusqu'à la fin de la ligne**
.	**répétition de la dernière commande**
U	**annulation de la dernière commande**
H	**déplacement vers la gauche**
L	**déplacement vers la droite**
J	**déplacement vers le bas**
K	**déplacement vers le haut**
CTRL F	**déplacement sur la page suivante**
CTRL B	**déplacement sur la page précédente**
:	**entrée dans le mode de commande pour les commandes suivantes**
w [fic]	**sauvegarde dans le/un fichier**
Q	**sortie de vi**
q!	**sortie sans sauvegarde**

X	**sortie avec sauvegarde**
r fic	**insertion de fic dans le fichier courant**
! cmd_du_shell	**exécution d'une commande shell**
%s/chaine1/chaine2/g	**substitution dans tout le fichier de chaine1 ou rexp par chaine2 (plusieurs fois par ligne)**
sx,y/chaine1/chaine2/	**substitution de x à y de chaine1 ou rexp par chaine2**
s4,./chaine1/chaine2/g	**substitution de la 4e ligne à la ligne courante de chaine1 ou rexp par chaine2 (plusieurs fois par ligne)**
s4,$/chaine1/chaine2/gc	**substitution de la 4e ligne à la fin de chaine1 ou rexp par chaine2 (plusieurs fois par ligne) avec confirmation (y,n,q)**
g/chaine/l	**liste des lignes contenant chaine**

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

Tout ce que vous avez toujours voulu savoir sur Unix sans jamais oser le demander.

-Vincent Lozano-2011

VII/ Les principales commandes d'Unix

Remarque : Toutes les commandes s'écrivent en minuscule.

Commande d'identification : who et who am i

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

ls

Sous Unix, la commande la plus utilisée est, sans aucun doute, *ls*. Elle liste les fichiers et répertoires de votre répertoire courant.

mkdir

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

Ma_machine>mkdir projet1

Ma_machine>mkdir -p projet2/src/new

cd

La commande *cd* permet de changer de répertoire. Le répertoire courant devient celui précisé.

*Ma_machine>**cd bin***

*Ma_machine>**cd bin/new***

*Ma_machine>**cd ../../src***

La variable d'environnement ***$HOME*** permet de définir de manière générique son répertoire personnel.

Remarque: Taper *cd $HOME* <=> ***cd ~ <=> cd***

A noter deux répertoires spéciaux, le répertoire courant (c'est-à-dire dans lequel vous êtes) représenté par le point. et le répertoire père représenté par le double point..

cd . Ne sert donc à rien !

cp

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

Ma_machine>cp fic1 fic2

Ma_machine>cp fic1 rep

Ma_machine>cp -R rep1 rep2

mv

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

Ma_machine>mv fic1 fic2 #(fic1 n'existe plus)

Ma_machine>mv fic1 rep #(fic1 existe, mais sous rep)

rm

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

*Ma_machine>rm -f fic**

Ma_machine>rm -r rep

Ma_machine>rm -rf rep

Attention: *rm*, supprime réellement le fichier et il n'y a pas de restauration possible après.
Attention également au *rm toto** mal écrit, un doigt qui fourche, qui glisse pour faire un ***rm toto *.*** avec un "blanc" entre toto et * ==> PERTE de tous les fichiers du répertoire!

grep

grep permet de rechercher les occurrences d'un mot ou d'un morceau de mot dans un fichier.

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

- ***-i*** pour ne pas tenir compte des majuscules/minuscules,
- ***-v*** pour ignorer un mot,
- ***-n*** pour avoir les numéros de ligne,
- ***-E*** pour les expressions régulières plus compliquées,
- ***-l*** pour lister seulement les fichiers contenant la chaîne recherchée,
- ***-s*** pour supprimer les messages d'erreurs.

Pour comprendre voici quelques exemples :

- **nombre de lignes en commentaire (commençant par !)**

  dans un code Fortran :

  grep "^!" prog.f | wc -l

- **recherche de STOP avec le numéro de la ligne dans tous les**

  sources :

  grep -n -i stop *.f*

- **liste de tous les fichiers qui n'ont pas "image" ou "son"**

  dans leur nom :

  ls | grep -vE "(image|son)"

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

tail

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

- Si vous souhaitez 25 lignes ajoutez *-n 25* à la commande *tail*
- Si votre code est en train de s'exécuter vous pouvez suivre l'évolution de vos fichiers d'output ou de log par cette même commande agrémentée d'un *-f*.

  tail -f output affichera "en direct live" votre fichier. Sans cette commande, vous seriez obligé d'éditer le fichier et de le recharger toutes les 10 secondes !

- Si vous souhaitez faire le contraire, avoir le début du fichier, utilisez *head output*

find

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

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

- cet exemple va chercher tous les noms des fichiers qui comportent la chaîne de caractères *exe1* dans les noms des fichiers du répertoire *rep1* ainsi que dans tous ses sous-répertoires. Attention, la recherche est récursive.

  *Ma_machine>**find rep1 -name "*exe1*" -print***
  /share/thot/users/rep1/exe1
  /share/thot/users/rep1/pexe11
  /share/thot/users/rep1/llexe12

- Si vous cherchez un include particulier (au-dessous ***signal.h***) sans avoir la moindre idée de sa localisation, partez de la racine / :
  find / -name "signal.h" -print

  Attention, cela risque de prendre un peu de temps surtout si vous avez des disques de plusieurs giga octets ! De plus si vous n'êtes pas root, un grand nombre de messages vont apparaître indiquant l'impossibilité de traverser certains répertoires.

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

- Encore plus fort, si vous voulez supprimer directement les fichiers trouvés, il est possible plutôt que de faire afficher seulement les noms de ces fichiers, de les détruire également :
  find ~ -name "core" -exec rm {} \;
  Remarque : ***{}*** permet de passer les noms des fichiers trouvés par find à la commande ***rm*** et **le** ***\;*** à la fin est obligatoire.
- Avec deux filtres c'est possible !
  find ~ \(-name "*.L" -o -name "*.o" \) -exec rm {} \;
  équivalent à
  find ~ -name "*.[Lo]" -exec rm {} \;
- En fait tout est possible, les filtres sont évalués de gauche à droite. Ici on essaye de supprimer les fichiers contenant "chaine" ; si on n'arrive pas à faire le rm, on affiche le fichier
  find . -exec grep -q chaine {} \; ! -exec rm {} \; -print
- Il est possible de spécifier dans le filtre le type d'élément cherché (*fichier* ***-type f***, ou répertoire ***-type d***) ou ayant une taille particulière (***-size nk***), le ***-ok*** a le même rôle que ***-exec*** avec la confirmation en plus.
  find . -type -d -print
  find . -size 0k -ok rm {} \;
- Il est également possible de spécifier la date de l'élément cherché (***-atime +n*** avec n le nombre de jours depuis le dernier accès)
  find / \(-name "a.out" -atime +7\) -o \(-name "*.o" -atime +30\) -exec rm {} \;
- Un autre cas pratique, vous n'arrivez pas à faire un ***rm*** sur un fichier car son nom contient un caractère "bizarre". Nous allons demander à ***find*** de détruire (avec confirmation via un ***-ok***) un fichier par son numéro d'inode.
  ls -i (pour trouver son numéro d'inode ici 733)
  find . -inum 733 -ok rm {} \;
  - Remarque sur le ***rm*** : si vous voulez supprimer un fichier qui commence par
    - **-** , vous allez avoir quelques surprises, alors utilisez l'option **--** de ***rm*** ainsi :
      rm -- fic_avec_un-
    - **# , faites *rm ./#fic_avec_un#* ou *rm \#fic_avec_un#***

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

Tableau I

Bash signifie « Bourne Again Shell » : un jeu de mot avec la construction anglaise « born again » qui signifie renaissance.

VIII/ Faire Connaissance avec le Shell.

a. Introduction au shell

Après avoir ouvert une fenêtre terminal, vous êtes pris en charge par un interpréteur de commandes : le Shell. Ce programme interactif invite l'utilisateur à introduire une ligne de commande(s), qui après vérification sera exécutée. Une fois terminée, le contrôle reviendra au shell qui relancera alors le dialogue.

Le shell bash (Bourne Again Shell) est l'un des shells les plus utilisés sur un vaste panel de plates-formes (Linux, Mac OS X, BSD et bien d'autres OS). Son succès tient principalement à sa grande souplesse.

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

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

Sur votre station ou votre PC sous Linux vous avez certainement le choix entre le ***bash,*** le ***ksh***, le ***tcsh*** Je conseille donc naturellement aux débutants le shell ***bash*** qui a été écrit pour Linux et possède un grand nombre de fonctionnalités communes.

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

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

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

b. Mecanismes essentiels du shell

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

***** : équivaux à "n'importe quoi" (chaîne, caractère ...) sauf. et **/**
ls *.c listera les fichiers ayant comme suffixe un c

? : remplace un caractère dans une chaîne
cat t?t? affichera le contenu de toto, titi, tata et t1t2

[group]: sélectionne certains caractères
cat t[ia]t[ia] affichera le contenu de titi et tata, mais pas toto ni t1t2

[début-fin]: sélectionne un intervalle
cat [0-9]*[!0-9] affichera le contenu de tous les fichiers ayant un chiffre au début mais sans chiffre à la fin (dû au !).

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

Remarque: avec la commande *echo* *il ne faut pas confondre / et \ en Unix.*

Trois (03) quotes à ne pas confondre :

Si vous voulez que rien ne soit interprété, placez des ' (quotes) de part et d'autre de votre chaîne.

>echo '$?*_\'

>$?*_

Si vous voulez quand même interpréter les dollars, utilisez les " (doubles quotes),

>echo "$PATH?*_\"

>/usr/bin?*_

Si vous voulez exécuter une chaîne, il est nécessaire de la placer entre ` (back quotes).

>REP=`ls -lrt *.rpm`;echo $REP

>-rw-r--r-- 1 jpp jpp 134226 nov 21 17:51 libcdf-2.7-1.i386.rpm

c. Quelques commandes Unix pour écrire des scripts

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

Quelques exemples de traitements "inhabituels" :

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

Mais avant de vous lancer dans l'écriture de scripts, soyez sûr que le traitement que vous souhaitez réaliser ne soit pas déjà effectué par une commande unix ! (ça arrive TRÈS souvent au débutant !). Aussi avant de voir plus en détail

l'implémentation des exemples cités plus haut, passons en revue quelques commandes unix puissantes encore non citées dans ce polycopié

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

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

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

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

paste : permet de "coller" deux fichiers l'un à côté de l'autre, c'est-à-dire d'ajouter le contenu de la ligne i du second fichier à la suite de la ligne i du premier fichier et cela pour toutes les lignes.
Ne pas confondre avec l'un derrière l'autre (*cat fic >> fic2*) qui met les lignes du fichier 2 derrière celles du fichier 1)

- *paste fic1 fic2 >fic3*
- *paste -d\| fic1 fic2 > fic3*
  pour mettre un | caractère de séparation entre la fin de la ligne du fichier 1 et avant le début de la ligne du fichier 2 et cela pour chaque ligne

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

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

tr : *tr b-z a-y*, sert à convertir le flux de caractères en remplaçant une lettre de la première liste par une autre de la deuxième liste (du même indice). Par

exemple *tr a-z A-Z* permet de passer toutes les lettres d'un flux en majuscules. Deux options intéressantes :

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

Voir la documentation avec la commande man.

d. L'interpréteur de commandes ou Shell bash

Remarque : Il ne faut pas mettre d'espace autour du signe =. Le shell ne comprendrait pas qu'il s'agit d'une affectation.

L'interpréteur de commandes UNIX , encore dit SHELL (coquille en anglais) lit le nom de la commande et tout ce qui suit sert d'arguments à la commande exceptés certains caractères spéciaux appelés métacaractères & < > * ? | \ qui doivent être interprétés.

Par exemple: ls -l * liste selon l'ordre alphabétique le nom de tous les fichiers et répertoires du répertoire courant en précisant la date de leur dernière utilisation, leur taille, leur état, ...

L'interpréteur exécute la commande en créant un nouveau processus, dit processus fils, et attend la fin de son exécution avant de permettre la frappe d'une autre commande. Il suffit d'ajouter en fin de commande le caractère & pour que cette attente par le processus père de la fin de l'exécution du processus fils n'ait pas lieu. Avec &, la commande est dite être exécutée en arrière-plan.

L'interpréteur permet aussi de ranger dans un fichier une suite, une procédure de commandes et de les faire exécuter en invoquant le nom du fichier suivi des paramètres ce qui transforme cette procédure en une nouvelle commande. Ainsi, il est possible de créer un environnement sur mesure, adapté aux besoins.

Comment écrire une procédure de commandes ou script?

Les règles du langage de commandes:

L'écriture d'une telle procédure dans le fichier de nom, par exemple: commande, doit suivre les règles suivantes :

- toute variable est de type chaîne de caractères (non encadrées d'apostrophes et non séparées par des virgules)
- des structures de contrôles peuvent être utilisées
- les paramètres sont supposés initialisés à l'appel
- la procédure hérite des variables d'environnement
- # en première colonne entraine que le reste de la ligne est un commentaire.

Remarques :

- Une exception, toutefois, au commentaire #, la première ligne d'une commande #!/bin/sh ou #!/bin/bash ou ... lance l'exécution d'un processus sh ou bash qui peut être différent du shell par défaut (bash).
- Le fichier, pour être exécutable, doit avoir les droits d'accès r-x (sinon exécuter: chmod +rx [ugoa] commande)
- Pour chaque exécution, un nouveau processus (fils) est utilisé

La désignation des paramètres :

À l'intérieur de la procédure de commandes

- - le nom de la commande, c'est à dire du fichier la contenant, est désignable par $0
- - le premier paramètre est désignable par $1
- - le second paramètre est désignable par $2 ...
- - la liste complète des paramètres est désignable par $*
- - le nombre de paramètres est désignable par $#
- - le numéro du processus en cours d'exécution est désignable par $$
- - le numéro du dernier processus exécuté avec & est désignable par $!
- - l'option de sh (-v ou -x ...) est désignable par $-
- - le code de retour d'une commande est désignable par $?

Exemple: Soit une procédure de commandes, stockée dans le fichier de nom chercher. Son appel sous la forme: chercher MOT1 MOT2 entraine que :

- $0 désigne chercher
- $1 désigne MOT1
- $2 désigne MOT2
- $* désigne MOT1 MOT2
- $# désigne 2

e. Manipulation des variables : locales et d'environnement

- Les variables locales :Elles n'ont pas à être déclarées. Il suffit de les initialiser. Par exemple:
  VarLocale1=valeur1 VarLocale2=valeur2 ... La valeur de VarLocale1 est $VarLocale1 ou ${VarLocale1} ce qui en permet la concaténation avec tout autre texte. Pour rendre ces variables locales permanentes, il faut les "exporter" par
  export VarLocale1 VarLocale2
  Dès lors, ces variables deviennent des variables d'environnement.

- **Les variables d'environnement:**

  La procédure de commandes, lors de son exécution, hérite de l'environnement, et notamment, des variables d'environnement. Pour en obtenir la liste, il suffit de taper env. Le résultat de l'exécution de env, obtenu sur le PC aube est du type suivant:

f. Les structures de contrôle

La boucle for:

Sa forme générale est

```
for var in Liste
do
   commandes avec $var et/ou les variables d'environnement
done
```

exécute commandes avec la première valeur de la liste, puis, la seconde, ... Derrière in, vous pouvez mettre entre ` ` n'importe quelle commande dès lors qu'elle génère une liste d'éléments du même ordre. *`cat fic1`* ou *`grep [0-9] fic1`* peuvent convenir ou simplement une liste comme *1 20 50*. Le traitement interne de la boucle peut être toute commande shell habituelle.

Exemple : **Le fichier boucle1 contient**

for i in $*
do
echo Le parametre est $i
done
et taper boucle1 a bb 45 donne boucle1: Permission denied

Il faut donner au moins les droits d'accès r-x par la frappe de chmod 500 boucle1
Alors, la frappe de boucle1 a bb 45 affiche :

Le parametre est a
Le parametre est bb
Le parametre est 45

Si le fichier boucle2 contient

for i in $*
do
echo 'Le parametre est $i'
done
alors la frappe de boucle2 a bb 45 donne :
Le parametre est $i
Le parametre est $i
Le parametre est $i
La mise entre apostrophes a empêché l'interprétation de $i qui deviennent alors 2 caractères standards. La mise entre apostrophes neutralise l'interprétation des $var.

Si le fichier boucle3 contient

for i in $*
do
*** echo "Le parametre est $i"***
done
alors la frappe de boucle3 a bb 45 donne :
Le parametre est a
Le parametre est bb
Le parametre est 45
$i est de nouveau interprété mais le caractère * entre les "" ne le serait pas.

Si le fichier boucle4 contient

for i in `ls $MEFISTO`
do
*** echo "Le parametre est $i"***
done
alors la frappe de boucle4 affiche :
Le parametre est bin
Le parametre est doc
Le parametre est elas
Le parametre est incl
...
Le parametre est xvue

Il est possible de créer une liste de paramètres à partir des résultats de l'exécution d'une commande obtenus en encadrant l'appel de la commande par le caractère `

Par exemple: rep=`pwd` echo $rep donne /share/thot/users/dea-ana/martin/mefistox

La réaction du shell pour un caractère spécial encadré par un même caractère spécial est la suivante :

f signifie fin de chaîne i signifie interprété n signifie non interprété

Le caractère à interpréter ou non	'	"	`	\	$	*
Entre ' le caractère ci-dessus vaut :	f	n	n	n	n	n
Entre " le caractère ci-dessus vaut :	n	f	i	i	i	n
Entre ` le caractère ci-dessus vaut :	n	n	f	i	n	n

Remarques :

- ***Pour écrire une commande sur plusieurs lignes, il faut terminer chaque ligne, sauf la dernière par \ et sans aucun blanc derrière ou autre caractère. En effet, \ neutralise le caractère \nl de fin de ligne ce qu'il ne réalise pas si un caractère est situé entre \ et \nl.***

Exemple :

```
for i in `ls $MEFISTO`
do
  echo \
  "Le parametre est $i"
done
```

donne le résultat précédent.

f.1 Les boucles conditionnelles while et until:

Sa forme générale est

```
while commandes1
do
    commandes2
done
```

et

```
until commandes1
do
   commandes2
done
```

Dans le premier cas, commandes2 est exécuté si le code de retour de commandes1 est 0 (pas d'erreur). Dans le second cas, commandes2 est exécuté si le code de retour de commandes1 est >0.

break / continue :

- break fait sortir de l'exécution de la boucle for ou while ou until
- break 3 fait sortir de l'exécution de 3 boucles for ou while ou until imbriquées
- continue fait passer à l'itération suivante de la boucle
- continue 4 fait poursuivre l'exécution de la boucle 4 itérations plus loin.

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

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

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

```
#!/bin/bash
ls |
{
   while read ligne
   do
     echo $ligne
   done
}
```

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

```
…
while commande1;do commande2;done
…
```

f.2 Le choix case:

```
case $Var in
   option1) commandes1 ;;
   option2) commandes2 ;;
   …
   optionm) commandesm ;;
   *) commandes ;;
esac
```

Si $Var vaut option i, alors, les commandes i sont exécutées, puis, la commande au-dessous de esac est exécutée. Le caractère * signifie "n'importe quel caractère" et cette option joue le rôle de sinon c'est à dire de tous les autres cas possibles.

Option peut être défini par
- plusieurs lettres entre [ ] ; par exemple: [abcd]) echo $Var ;;
- plusieurs options avec le sens de OU ; par exemple: -a|-b|-c) echo $Var ;;
- pour neutraliser un caractère spécial, il faut le précéder de \

Exemple:

```
case $REP in
  [hH]elp  ) echo j'ai tapé help;;
  go|run) echo le code démarre;;
  *     ) echo le reste;;
esac
```

f.3 Le test if:

Sa forme générale est

```
if commandes1
then
  commandes2
else
  commandes3
fi
commande4
```

Si le résultat de commandes1 (si plusieurs commandes sont exécutées alors le code efficace est celui de la dernière commande exécutée) est 0 (qui joue le rôle de vrai c'est à dire sans erreur d'exécution), alors, commandes2 sont exécutées, puis, commande4.

Si le résultat de commandes1 est différent de 0, alors les commandes3 sont exécutées, puis commande4. Cette option else est facultative.

Exemple:

```
cd
if test -d $MEFISTO
  ls $MEFISTO
else
  echo Pas de repertoire $MEFISTO
fi
```

donne

resultat: bin doc elas incl mail ot poba pp reso td test ther util xvue

De même,

```
if test -f $1
then
  :
else
  >$1
fi
```

exécuté sous la forme

test2 fichier

ls -l fichier

donne **-rw-r--r-- 1 martin dea-ana 0 sept 5 13:22 fichier**

Le fichier de nom fichier a été créé vide car il n'existait pas dans le répertoire courant.

Le caractère : est une commande vide retournant le code d'erreur 0 . Il permet ici de traiter la négation du test.

test -d rep retourne 0 si le répertoire rep existe, un entier >0 sinon

test -f fic retourne 0 si le fichier fic existe, un entier >0 sinon

test ! -f fic retourne un entier >0 le fichier fic existe, 0 sinon

test -r fic retourne 0 si le fichier fic existe et est ouvert en lecture, un entier >0 sinon

test -w fic retourne 0 si le fichier fic existe et est ouvert en écriture, un entier >0 sinon

test var retourne 0 si var n'est pas la chaîne vide, un entier >0 sinon

```
if commandes1
then
  commandes2
else
  if commandes3
  then
    commandes4
  fi
fi
```

peut être écrit sous la forme équivalente

```
if commandes1
then
  commandes2
elif commandes3
  commandes4
fi
```

Exemple:

```
echo "Taper OUI ou NON : O/N"
read var
if [ $var = o ] || [ $var = O ]
then
```

```
  echo OUI
else
  echo NON
fi
```

Les tests peuvent porter aussi sur les chaînes ou l'état des fichiers avec [[ ]], ou sur les nombres avec (( ))

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

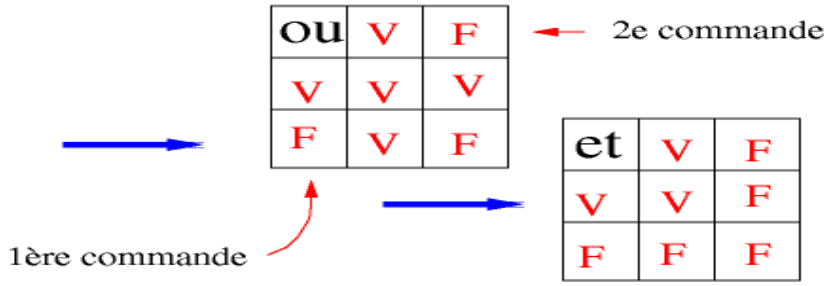
ou

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

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

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

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



Dans le cas d'un *ou*,

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

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

commandes1 && commandes2
exécute commandes2 seulement si le code d'erreur de commandes1 est 0 ce qui équivaut au test if sans la clause else.

commandes1 || commandes2
exécute commandes2 seulement si le code d'erreur de commandes1 est >0

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

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

g. Quelques commandes supplémentaires

Segmentation du résultat d'une commande en plusieurs paramètres set:

g.1 set `date` ; echo $*; echo $6 $5 $4 $3 $2 $1
donne

```
mar sep 10 17:40:42 CEST 2002
2002 CEST 17:40:42 10 sep mar
```

Le texte résultat a été segmenté en 6 paramètres qui ont ensuite été affichés en sens inverse.

Compléments sur la commande set :

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

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

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

g.2 Manipulation des expressions arithmétiques avec expr:

La frappe de i=1
i=`expr $i + 1`
echo $i
donne
2

Les opérateurs logiques et arithmétiques standards sont utilisables. Attention, les opérateurs doivent être encadrés de caractères blancs et les caractères spéciaux (>, >= , <= , < , | , & , * , $) précédés du caractère \ pour ne pas être confondus avec le caractère spécial. Ils sont donnés ici dans l'ordre décroissant de leur priorité:

Ici, expr, expr1 et expr2 représentent chacun une expression.

(expr) le groupe le plus interne est calculé en premier
expr1 % expr2 donne le reste de la division de expr1 par expr2
expr1 * expr2 et à égalité de priorité expr1 / expr2
expr1 + expr2 et à égalité de priorité expr1 - expr2

Si comp est l'un des opérateurs < , <= , = , != , >= , > alors, expr1 comp expr2 donne la valeur 1 si l'opération de comparaison numérique entre valeurs numériques ou lexicographique entre chaînes de caractères est vraie.

expr1 & expr2 donne la valeur de expr1 si expr1 et expr2 sont non nulles, sinon celle de expr2. expr1 | expr2 donne la valeur de expr1 si elle est non nulle, sinon celle de expr2.

g.3 exit arrête l'exécution et donne le code de retour de la dernière commande exécutée. exit 5 arrête l'exécution et donne le code de retour 5.

g.4 shift décale les paramètres d'un cran, $2 devient $1, $3 devient $2, ...

g.5 time commande donne le temps
- écoulé depuis le début de l'exécution de la commande
- consommé par l'unité centrale
- consommé par l'exécution de la commande.

Exemple: time `ls` affiche le contenu du répertoire courant puis {\tenttbis real 0m0.34s user 0m0.02s sys 0m0.04s}

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

...

Commande Compliquée

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

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

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

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

- *echo $0* : pour le nom de la commande
- *echo $n* : pour le paramètre de rang n ; il est possible de les décaler avec *shift* (sauf $0)
- *echo $$* : pour le numéro du processus courant (très utile pour créer des fichiers temporaires */tmp/poub_$$*)
- (( ))
- pour le calcul de variables :
  comme nous l'avons déjà vu, l'utilisation des ((et)) autour de l'expression

numérique permet de l'évaluer. Les opérateurs sont ceux habituellement utilisés dans d'autres langages : / % - + * (% pour le reste de la division euclidienne).
echo mon resultat est : $((354%54))

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

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

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

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

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

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

g.6 Vous pouvez également définir des tableaux à une dimension ainsi :

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

ou plus synthétiquement :

set +A nom_du_tableau champ1 champ2 champ3

Pour utiliser un tableau attention aux *{}* :

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

premier

g.7 La commande *eval* permet d'évaluer une ligne de commande :

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

g.8 Fonctions :

Comme dans de nombreux langages, vous pouvez créer vos propres fonctions. La syntaxe est la suivante :
nom_de_la_fonction () { liste; }

le blanc entre { et avant liste et le point virgule après liste sont obligatoires. Le code de retour est celui de la dernière commande exécutée dans la fonction. Il n'y a pas de nouveau *process* créé. De plus toutes les variables de la fonction sont partagées sauf si l'attribut "local" est précisé.

local nom_de_la_variable_locale=456

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

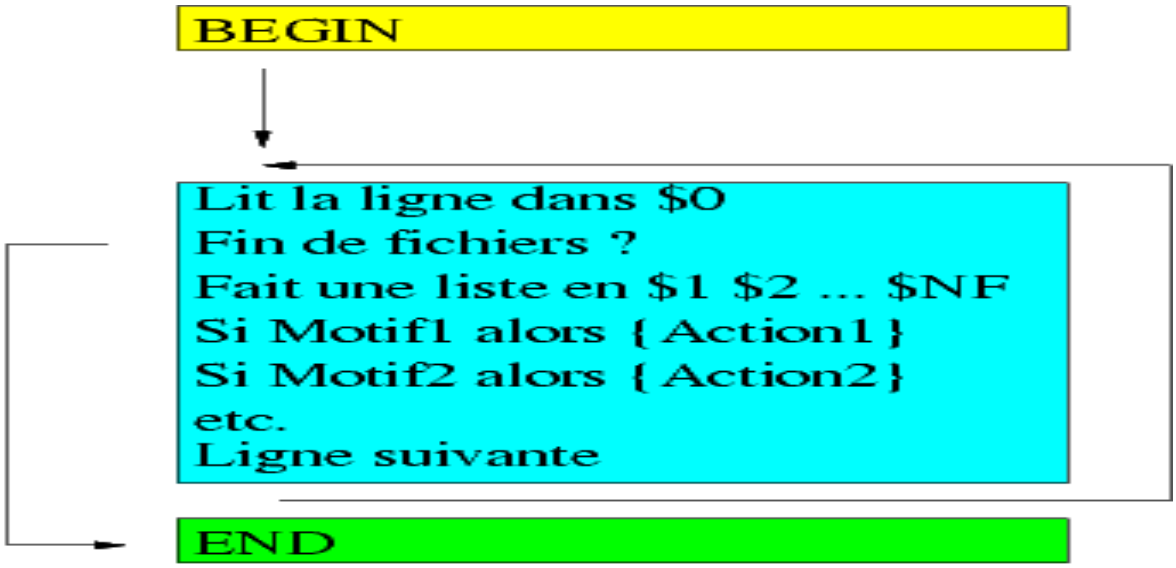
Remarque : ne jamais nommer son script "test", c'est une commande déjà vue dans if de Unix!

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

g.9 La commande awk :

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

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



Quelques exemples pratiques :

- Affiche les 2e et 4e champs d'un fichier fic1 (pour chaque ligne) avec un format
  awk '{printf ("%-9s | %3s \n",$2,$4)}' fic1
- Affiche les champs 3 et 2 si le champ 1 contient *URGENT* et compte les occurrences
  awk 'BEGIN{x=0} $1 ~ /URGENT/ {print $3, $2 ; ++x} END{print "total" x}' fic1
- Changement du séparateur de champs par défaut (" " ou TAB)
  awk 'BEGIN{FS=":"} print $1; print "la ligne complète est" $0' fic1
- Analyse des couleurs d'un document html (version 1)
  grep color unix_u_cours.html | awk 'BEGIN{FS="#"}{ print $2}' | cut -c1-6 | sort | uniq -c
- Analyse des couleurs d'un document html (version 2)
  awk 'BEGIN{FS="#"} $0 ~ /color/ {print substr($2,0,6) } ' unix_u_cours.html | sort | uniq -c
- Somme des éléments d'un fichier poub
  awk 'BEGIN{a=0} {a=a+$0} END{print a}' poub
- Moyenne des éléments communs d'un fichier poub (nom chiffre)
  awk 'BEGIN{NOM="";compteur=0;tot=0} { if ($1 == NOM) {tot=tot+$2;compteur=compteur+1} else { if (NR!=1) print tot/compteur,NOM ; NOM=$1;tot=$2;compteur=1} }END{print tot/compteur,NOM } ' poub

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

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

Cas des structures de commandes :

- les décisions (if, else) dans un bloc de commande
  awk '{if ($2=="") { print $3} else {print $2} } END {print "fini"}' fic1
- les décisions en dehors d'un bloc de commande
  awk 'NF>=7 { print $3 }' fic1
  awk '$2 ~ /toto/ || $1=="URGENT" { print $3 }' fic1
- les boucles (while, for, do-while)
  awk '{i=4;while (i<=NF) { print $i; i++} }' fic1
  awk '{for (i=11;i>=0;i--) { print $i} }' fic1

Il est possible de créer des tableaux :

- mono-dimensionnés :
  Tab[87]="milieu"
- associatifs :
  Tab["rouge"]=924

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

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

IX/ Les redirections: gestion des flux et des processus

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

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

a/ Série d'exemples :

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

    who >> mon_premier_fichier

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

- **Exemple de lecture d'un fichier avec < :**
  - Si vous avez un script *mon_script* qui vous pose de multiples questions, vous pouvez l'automatiser grâce au <. Mettez les réponses dans le fichier *les_reponses* et faites comme suit : ***mon_script < les_reponses***
  - Avec les deux symboles :
    mon_script < les_reponses > les_sorties
  - Avec deux < on envoie à la commande toutes les lignes tapées en entrée, jusqu'à la chaîne fin, méthode du "here document" :
    wc -l << fin

"collez quelque chose avec la souris"
fin

- Encore avec deux < on envoie à la commande toutes les lignes tapées en entrée, jusqu'à la chaîne fin mais cette fois-ci dans un fichier:

  cat > fic << fin
  "collez quelque chose avec la souris ou tapez du code ..."
  fin

- **Exemple d'utilisation d'un pipe | :**

  - Si l'on cherche à connaître le nombre de fichiers d'un répertoire :

    ls -la | wc -l

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

    commande_bavarde | more

  - Le pipe sert très souvent à remplacer le fichier input attendu dans la plupart des commandes par un flux de la commande précédente

    cat * | sort
    à la place de
    sort *

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

- Pour fusionner les deux flux (sortie et erreur), utilisez cette syntaxe : ***commande 1> poub 2>&1*** qui redirigera le flux 2 vers le flux 1 (attention uniquement avec la famille sh).
- Pour rediriger la sortie d'erreur dans un fichier (sans bloquer le flux standard):

  commande 2>erreur

- Si vous voulez que les sorties ne soient pas affichées ou conservées dans un fichier, il suffit de les envoyer vers ***/dev/null*** ainsi :
  script_bavard > /dev/null
- Pour rediriger la sortie d'erreur dans un pipe sans le flux standard :
  (commande 1>/dev/null) 2>&1 | more

b/ Caractéristiques des processus :

● Processus fils / père :

– ex : Le shell est un processus comme les autres Chaque commande exécutée correspond à la création d'un processus « fils » par rapport au shell (« père »)

Sous Unix, chaque processus est identifié par :

» PID (Processus Identifier) ; il est unique.

» PPID (Parent Processus Identifier)

Deux (2) types de processus :

- ✓ processus systèmes (daemons) : Exécution de tâches générales, souvent contrôlées par root
- ✓ processus utilisateurs

Sous UNIX, voici une petite astuce permettant de récupérer l'identifiant du processus (PID) ouvert par un script bash.

$$ récupère le pid de script courant

$! récupère le pid du dernier script lancé

Ainsi, il est possible de contrôler l'exécution d'une sous tâche grâce au PID du processus. Pour vous en convaincre, un petit exemple de script BASH :

```
#!/bin/bash

echo "PID du processus courant : $$"

echo "PID du processus lancé : $!"

# On lance une commande en background avec => &

# Ca peut être une commande, un autre script bash, un programme, etc...

ping localhost &
```

```
echo "PID du processus courant : $$"

echo "PID du processus lancé (ping localhost) : $!"

sleep 3

kill $!

exit 0
```

Tout comme les fichiers, un système UNIX identifie les processus grâce à un nombre. Cette information s'appelle le PID (Process IDentificator). Comme un fichier, un processus a un propriétaire. L'analogie s'arrête ici. Toutes les informations concernant les processus s'obtiennent avec la commande ps. (Vous pouvez retrouver l'ensemble des options en consultant le manuel en ligne).

En tapant la commande ps axl, vous allez obtenir des informations concernant tous les processus :

– le propriétaire (UID) ;

– le numéro du processus (PID) ;

– le numéro du processus père (PPID) ;

– la priorité (PRI) ;

– la valeur de NICE (NI) ;

– l'état (STAT)

– le nom du device vers lequel il est dirigé (TTY) ;

– le temps passé dans la cpu (TIME) ;

– le nom de la commande exécutée (COMMAND).

Si vous ajoutez l'option "f, les liens de parenté entre les différents processus apparaitront plus facilement. Notez que tous les processus sont lancés par le processus init dont le PID est 1. Le processus père du processus init n'est autre que le noyau (PID ´égal à 0). Vous pouvez utiliser la commande pstree pour visualiser d'une seconde manière l'arborescence des processus.

Les états d'un processus peuvent être les suivants :

– D : le processus est in interruptible ;

– R : le processus est en train de s'exécuter ;

– S : le processus est endormi ;

– T : le processus est stoppé ;

– Z : le processus est zombi : il est mort mais son père ne le sait pas.

c/ Processus exécuté en avant ou arrière-plan :

Lorsque vous utilisez des commandes ou vos programmes favoris sous Unix, ils s'exécutent systématiquement sous votre shell.

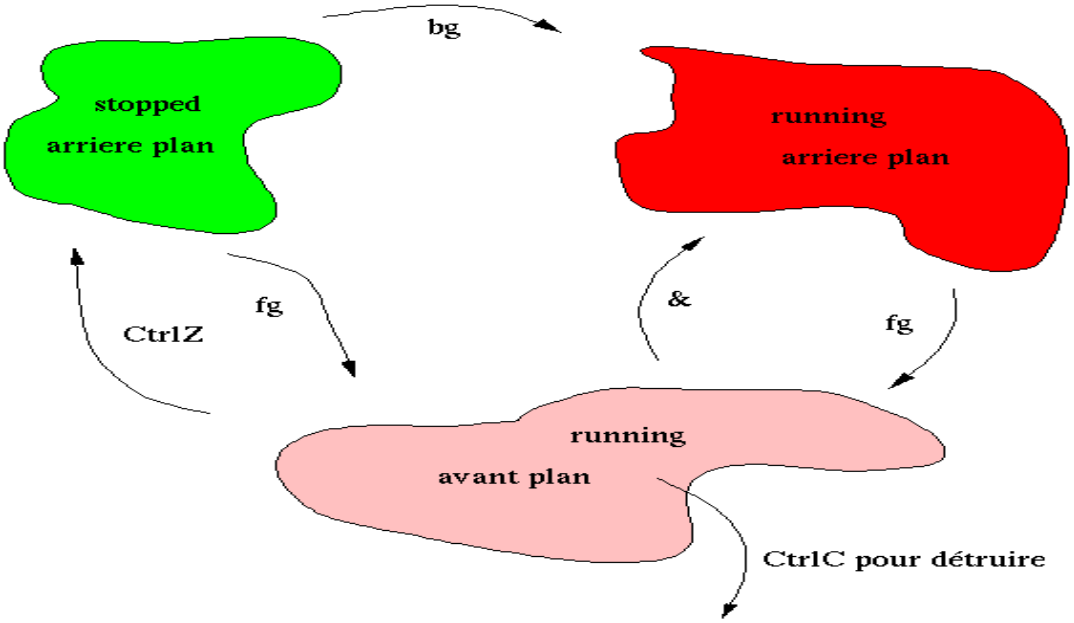
Tant que la commande s'exécute, vous n'avez plus la main, vous êtes bloqué. Certains utilisent plusieurs fenêtres pour continuer à travailler.

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

CTRL-z / bg / fg

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

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



- **ps /kill**

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

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

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

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

wait fait attendre le processus père jusqu'à la fin de l'exécution de tous ses processus fils wait 1034 fait attendre jusqu'à la fin de l'exécution du processus 1034.

d/ Ecriture de quelques scripts simples en shell (bourne –shell) :

01/ Exemple en utilisant l'éditeur vi suivi du nom du script : vi essai

```
#!/bin/bash
a=1
let b=2
echo $a $b
```

Après l'exécution de ce script du nom essai : sh essai ; il affiche 1 2

02/ Exemple :

```
#!/bin/bash
echo "entrez votre nom"
read nom
echo "vous vous appelez $nom"
echo "entrez le prenom"
read prenom
echo "votre prenom $prenom "
echo "votre date de naiss"
read naiss
echo "vous êtes  nez $naiss"
```

03/ Exemple :

```
#!/bin/bash
read var
if [ $var -eq 5 ]
then
 echo "var est egal a 5";
else
 echo "var ne vaut pas 5";
fi
```

04/ Exemple avec la boucle for:

```
#!/bin/bash
for i in 1 2 3 4 5 ; do
echo "bonjour $i fois "
done
```

05/ Exemple : obtenir le PID d'un ou de plusieurs processus en arrière-plan:

```
#!/bin/bash

xcalc &

d=$!

echo " votre pid $d "

wait $!

xclock &

f=$!

echo " votre pid $f "
wait $!
xterm &
c=$!
echo " votre pid $c "
```

06/ Exemple utilisant le case :

```
#!/bin/bash
echo " si vous voulez quittez tapez le mot exit "
while true
do
if [ $var =  "exit"  ]
then
 exit 0
fi
echo " entrez charctere "
read var
case $var in
[A-Z])
echo " une lettre maj "
;;
[a-z])
 echo "  une lettre miniscule "
 ;;
[0-9])
 echo " un nombre "
```

```
 ;;
?) echo " caractere special "
  ;;
*)
 echo " vous avez entré plus d'un caractere "
;;
esac
done
```

e/ Ecriture de quelques scripts traduits en différents shell (bourne - shell, korn –shell, C-shell, T-Cshell) :

01/ Ecriture du Script en shell (bourne shell)

```
#!/bin/bash
echo "Bienvenue ce programme va afficher la liste des
processus lancés par l'utilisateur et afficher leur état
respectif."

if ( test $# -eq 0 )
then echo ' Erreur syntaxe'
else ps hU $1 -o state,command > tempfile
number=$(wc -l < tempfile)
i=1
while [ $i -le $number ]
do
commande=$(tail -$i tempfile|head -1|cut -d" " -f2)
etat=$(tail -$i tempfile|head -1|cut -d" " -f1)
i=$(( i+1 ))
echo -n " $commande :"
case $etat
in
D) echo "endormi  => ininterruptible  D" ;;
S) echo "endormi S" ;;
R) echo "en cours R" ;;
T) echo "stop T" ;;
Z) echo "zombi Z" ;;
*) echo "inconnu" ;;
esac
done
rm tempfile
fi
```

Exécutions :
Bourshell : sh suivi du nom du fichier
Korn-shell ksh suivi du nom du fichier

02/ Ecriture du Script en C-shell

```
#!/bin/csh

echo "Bienvenue ce programme va afficher la liste des processus lancés par l'utilisateur et afficher leur état respectif."

while (! $?listProcess )

echo "Entrer un nom d'utilisateur : "
set userName=$<
set isExist=`id -u $userName`

if ("$isExist" != "") then
set listProcess=`ps --no-headers -o stat,cmd -u $userName >> outEXO`

endif

end

echo ""

cat outEXO | awk '{ \
if($1=="S")        printf"ETAT: endormi -20sec                    "; \
else if($1=="Ss")   printf"ETAT: endormi                          "; \
else if($1=="I")    printf"ETAT: endormi +20sec                   "; \
else if($1=="R")    printf"ETAT: en execution                     "; \
else if($1=="Z")    printf"ETAT: zombi                            "; \
else if($1=="T")    printf"ETAT: suspendu                         "; \
else if($1=="D")    printf"ETAT: endormi instoppable              "; \
else if($1=="X")    printf"ETAT: mort                             "; \
else if($1=="Sl")   printf"ETAT: endormi multi-threaded           "; \
else if($1=="SN")   printf"ETAT: endormi Basse Priorité           "; \
else if($1=="S+")   printf"ETAT: endormi fg                       "; \
else               printf"ETAT: %s                              ",$1; \
printf"CMD: %s\n", $2; \
}'

rm outEXO
```

03/ Exemple : copie d'un fichier dans un autre fichier : simulation de la commande unix cp fichier1 fichier2 en langage C-shell

```
#!/bin/csh

set continuer = "oui"
set fichier1=$1
set fichier2=$2
while ( $continuer == "oui" || $continuer == "0UI" )
while ( ! -e "$fichier1" )
      echo "fichier source n'existe pas"
      echo "entrer le entrer le nom de fichier a copier: "
      set fichier1 = $<
end
if ( $# < 2 ) then
       echo "entrer le nom de la copie:"
       set fichier2 = $<
endif
set existe=0
while ( $existe == 0 )
  echo -n "donnez le repertoire:"
  set rep = $<
  if (  -d "$rep" ) then
  @ existe = 1
  else
   echo "ERREUR: le repertoire n'existe pas"
  endif
end
set repsource=`pwd`
cd $rep
if ( -e "$fichier2" ) then
   fichier2="$fichier2.`date "+%m%d_%H%M%S"`"
   echo "le fichier existe, le nouveau fichier sera: $fichier2"
endif
cd $repsource
cp $fichier1 "$rep/$fichier2"
      echo "vouler vous continuer:(OUI,oui/NON,non) "
      set continuer = $<
      while ( $continuer != "OUI" && $continuer != "oui" &&
$continuer != "NON" && $continuer != "non" )
       echo "ERREUR: entrer OUI,oui/NON,non"
       set continuer = $<
```

```
    end
    if ( $continuer == "oui" || $continuer == "0UI" )   then
      echo "entrer le nom de fichier a copier:"
      set fichier1 = $<

      echo "entrer le nom de la copie:"
      set fichier2 = $<
    endif
end
```

Exécutions :
C-shell : csh suivi du nom du fichier
T-Cshell : tcsh suivi du nom du fichier

04/ Simulation de la commande unix kill (suppression de différends processus) en script shell et c-shell

```
#!/bin/bash
ps u
echo -n "Quel processus voulez-vous supprimer: "
read name

PID=$(ps -Af | grep $USER | grep " $name" | grep -v grep | tr -s ' '
| cut -f 2 -d ' ')
if [  $(echo $PID | wc -c ) -eq 1 ]
then
      echo "pas de processus $name"
      exit 1
fi

echo $PID | sed 's/ /\n/g' > mytmp
n=$(cat mytmp | wc -l)
i=1
while [ $i -le $n ]
do
      currentPID=$(echo $PID | cut -f $i -d ' ' )
      if [ $currentPID -eq $$ ]
      then
            ps aux | grep "$currentPID" | grep -v grep
            exit 0
      fi
            ps aux | grep "$currentPID" | grep -v grep
            kill -9 $currentPID
```

```
        i=$(expr $i + 1)
done

#!/bin/csh
ps -u
echo -n "Quel processus voulez-vous supprimer: "
set name = $<


set PID=`ps -Af | grep $USER | grep " $name" | grep -v grep | tr -s
' ' | cut -f 2 -d ' ' `
if ((  `echo $PID | wc -c `  == 0 ))then
        echo "pas de processus $name"
        exit 1
endif

echo $PID | sed 's/ /\n/g' > mytmp
set n=`cat mytmp | wc -l`
set i=1
while ( $i <= $n )
        set currentPID=`echo $PID | cut -f $i -d ' ' `
        # remove this if (seulement si ca cause des problemes runtime
...)
        if (( $currentPID == $$ ))then
                ps aux | grep "$currentPID" | grep -v grep
                exit 0
        endif
                echo $currentPID
                kill -9 $currentPID
        @ i++
End
```

05/ Exemple complet

```
#Ecrire un script pere qio aura en charge le déclenchement
#de deux scripts enfants.
#Le script enfant1 écrit sans arrêt sur la console le mot ping
#Le script enfant2 écrit sans arrêt sur la console le mot pong
#Le script père fonctionne ainsi :
#    *
#     Déclenche le processus enfant1
#     Attend (wait 10)
#     Tue le processus enfant1
```

#
 # Déclenche le processus enfant2
#
 # Attend (wait 10)
#
 # Tue le processus enfant2#

```
#!/bin/bash
lancer() {
./$1.sh &
sleep 1
kill -9 `cat $1.pid` 2> /dev/null
rm $1.pid
}
while true; do
        lancer ping
        lancer pong
done
#Contenu du script enfant1 (ping.sh) :
#!/bin/bash
if [ -e *.pid ]; then sleep 1; fi
echo $$ > ping.pid
while true; do
        echo -n "ping "
        sleep 5
done
#Contenu du script enfant2 (pong.sh) :
#!/bin/bash
if [ -e *.pid ]; then sleep 1; fi
echo $$ > pong.pid
while true; do
        echo -n "pong"
        sleep 5
done
```

X/ Langage C Sous Unix

a/ Rappel du langage C avec des exemples

Le compilateur C sous UNIX s'appelle cc. On utilisera de préférence le compilateur gcc du projet GNU. GNU Compiler Collection.
gcc masource.c # crée un exécutable du nom de **a.out**, que l'on lancera avec **./a.out**.

gcc -o <nom_du_programme_que_l'on_souhaite_donner> -c <masource>.c # L'argument **-o** vous permet de choisir le nom de l'exécutable qui naîtra de cette compilation.
Ce compilateur est livré gratuitement avec sa documentation et ses sources. Par défaut, gcc active toutes les étapes de la compilation. On le lance par la commande

gcc [*options*] fichier.c [-l*librairies*]

En C, toute variable doit faire l'objet d'une déclaration avant d'être utilisée.

Un programme C se présente de la façon suivante :
main()
{*déclarations de variables internes*
instructions
}
Exemple

```
#include

main()
{
  int a = 0;
  int b = 1;
  if (a = b)
    printf("\n a et b sont egaux \n");
  else
    printf("\n a et b sont differents \n");
}
```

imprime à l'écran a et b sont egaux !

01/ Ce programme a pour seul but d'afficher le texte « Bonjour ! » suivi d'un retour à la ligne :

```
#include <stdio.h> /*BIBIOTHEQUE DES ENTREES ET SORTIES*/

 int main(void)
 {
    printf("Bonjour !\n");
    return 0;
 }
```

02/ Exemple avec des valeurs réelles :

```
main()
{ int i = 3, j = 2;
printf("%f \n",(float)i/j);
}
```

Apres compilation et exécution : retourne la valeur 1.5.

03/ Exemple d'extraction de la racine carrée d'un nombre :

```
#include <stdio.h>

double SquareRoot(double X) {
double A,B,M,XN;
if(X==0.0) {
 return 0.0;
} else {
  M=1.0;
        XN=X;
       while(XN>=2.0) {
          XN=0.25*XN;
          M=2.0*M;
       }
       while(XN<0.5) {
          XN=4.0*XN;
          M=0.5*M;
       }
       A=XN;
       B=1.0-XN;
       do {
          A=A*(1.0+0.5*B);
          B=0.25*(3.0+B)*B*B;
       } while(B>=1.0E-15);
       return A*M;
    }
  }

int main() {
      double R;
      printf ("introduire R");
      scanf ("%f",&R);
      while(R<=1000.0) {
```

```
        printf("Sqrt(%f)=%f\n",R,SquareRoot(R));
        R*=R;
    }
  }
```

04/ Exemple utilisant les tableaux : Triangle de Pascal :

```
#include<stdio.h>
const int nmax=100;
int main(){
int tab[nmax][nmax];
int n=-1;
int i,j;
char r='o';
do{
printf("Taper le degre du triangle:");
scanf("%d",&n);
for(i=0;i<=n+1;i++)
{
tab[i][i]=1;
tab[i][0];
for(j=1;j<i;j++)
tab[i][j]=(tab[i-1][j])+(tab[i-1][j-1]);
}
printf("triangle de pascal de degre %d: \n",n);
for(i=1;i<=n+1;i++){
printf("n=%d ",i-1);
for(j=0;j<=i;j++)
        if ((tab[i][j])!=0)
                        printf("%d ",tab[i][j]);
        printf("\n");
}
printf("Voulez vous continuer? (o ou n): ");
scanf("%s",&r);
} while(r=='o' &&  r=='O' || (r!='n'&& r!='N'));
return 0;
}
```

05/ Exemple de permutation :

sans test d'arrêt

```
#include<stdio.h>
main()
{
	int A,B,C,AIDE;
	printf("Introduisez trois nombres (A,B,C:");
	scanf("%i%i%i",&A,&B,&C);
	/*Affichage a l`aide de tabulations*/
	printf("A=%i\tB=%i\tC=%i\n",A,B,C);
	AIDE=A;
	A=C;
	C=B;
	B=AIDE;
	printf("A=%i\tB=%i\tC=%i\n",A,B,C);
	return 0;
}
```

06/ Exemple de permutation :

avec test d'arrêt

```
#include <stdio.h>
int main()
{
int A ,B ,C ,x ;
char choix = 'i' ;
while(choix != 'f')
{
printf(" ***** faites entrer les nombres : *****\n ") ;
printf(" A=  ") ;   scanf("%d",&A) ;
printf(" B= ") ;   scanf("%d", &B) ;
printf(" C= ") ;   scanf("%d", &C) ;
printf("\n");
 printf(" ****** les valeurs apres la permutation : ******\n");
x= B;
B=A ;
A=C ;
C=x ;
printf(" A= %d ",A);
printf(" B= %d ",B);
printf(" C= %d ",C) ;
  printf("\n");
```

```
 printf(" ***** Pour finir le programme taper f  : ");
  scanf("%s",&choix);
 printf("\n") ;
}
}
```

07/ Exemple du factoriel d'un nombre :

```
#include<stdio.h>
int main ()
{
int N,I;
double FACT;
do
{ printf("Entrez un entier:");
 scanf("%d",&N);
}
while(N<0);
/*a*/
/*I=1;
FACT=1;
while(I<=N)
      {
      FACT*=I;
      I++;
      }
*/
/*b*/

for (FACT=1.0,I=1;I<=N;I++)
      FACT*=I;

printf("%d!=%.0f\n",N,FACT);
return 0;
}
```

08/ Création des processus avec la fonction FORK :

Définition de la fonction FORK :

La fonction ***fork*** fait partie des appels système standard d'UNIX .

Cette fonction permet à un processus (un programme en cours d'exécution) de donner naissance à un nouveau processus qui est sa copie conforme, par

exemple en vue de réaliser un second traitement parallèlement au premier. Un bon moyen de visualiser l'effet d'un fork sur un processus est d'imaginer une division cellulaire.

Il existe une filiation dans les processus : le créateur d'un nouveau processus est appelé le père et le nouveau processus, le fils. La plupart des attributs système du père (par exemple les droits sur le système de fichier) sont transmis au fils, de la même manière que l'héritage. Au démarrage d'un système Unix, un seul processus existe (de numéro 1). Tous les autres processus qui peuvent exister au cours de la vie du système descendent de ce premier processus, appelé init, via des appels système comme fork[2], vfork[3], posix_spawn[4] ou d'autres moyens. Sur les premiers UNIX, seul l'appel système fork permet de créer de nouveaux processus[5,6].

```
//T0 un seul processus
f1=fork();
// T1 2 processus
f2=fork();
// T2 4 processus
f3=fork();
// T3 8 processus
/*
| T0 un seul processus
|\
| \
|  \
|   \ 1er fork
|    \
|     \
|      \
|       | T1 2 processus
|\      |\
| \     | \ 2eme fork
|  \    |  \
|   |   |   | T2 4 processus
|\  |\  |\  |\ 3e fork
| | | | | | | | T4  8 processus
*/
```

08-a Exemple

```
/*
 * fork-ex1.c : exemple d'utilisation de l'appel systeme fork() pour creer
 * un nouveau processus.
 */

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
```

```
int main (){
  pid_t valeur;
  printf("printf-0 Avant fork : ici processus numero %d\n",
(int)getpid());
  valeur = fork();
  printf("printf-1 Valeur retournee par la fonction fork: %d\n",
(int)valeur);
  printf("printf-2 Je suis le processus numero %d\n", (int)getpid());
  return 0;
}
```

08-b Exemple

```
/*
 * fork-ex2.c : exemple plus completd'utilisation de l'appel
systeme fork() pour creer
 * un nouveau processus.
 */

#include <stdio.h>
#include <errno.h>
#include <sys/types.h>


main()
{

  extern int errno;
  extern char *sys_errlist[];

  pid_t pid;

  pid = getpid();
  fprintf(stdout,"Avant le fork, pid = %d\n",pid);
  sleep(20);

  pid = fork();
  /* En cas de reussite du fork, le pere et le fils poursuivent leur
     execution a partir d'ici. La valeur de pid permet de distinguer
     le pere du fils.
  */
```

```
  switch(pid) {

  case -1:   /* erreur dans fork() */
    fprintf(stderr,"error %d in fork: %s\n",errno,sys_errlist[errno]);
    exit(errno);

  case 0:    /* on est dans le fils */
    fprintf(stdout,"Dans le fils, pid = %d\n",getpid());
    sleep(20);
    /* On suspend le processus pendant 20 secondes. Cela permet
d'utiliser la commande ps (p.ex. ps -gux) pour visualiser la liste des
processus.
    */
    break;

  default:   /* on est dans le pere */
    fprintf(stdout,"Dans le pere, pid = %d\n",getpid());
    sleep(20);
  }
}
```

b/ Traductions de quelques exemples du compilateur langage C vers l'interpréteur shell.

01/ Exemple :

programme en C qui parcourt l'ensemble des processus de l'utilisateur passé en paramètre et qui affiche leur état respectif.

a/ en langage C

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<errno.h>

int main(){

system("ps -u");
int test,i=0;
char STAT,user;
printf("entrer le nom  user\n");
scanf("%c",&user);
```

```
if(user==' '){
printf("erreur syntax:etat processus-user");}
else
{
switch(STAT){
case 'D' :
printf("endormi=>ininTERRUPTIBLE");

break;
case 'T':
printf("bloque");
break;
case 'S':
printf("endormi -20s");
break;
case 'I':
printf("endormi +20s");
break;
default :
printf("inconnu");
}
}
}
```

b/ Sa traduction en script Shell

```
 #!/bin/bash

if ( test $# -eq 0 )
then echo ' Erreur syntx'
else ps hU $1 -o state,command > tempfile
number=$(wc -l < tempfile)
i=1
while [ $i -le $number ]
do
commande=$(tail -$i tempfile|head -1|cut -d" " -f2)
etat=$(tail -$i tempfile|head -1|cut -d" " -f1)
i=$(( i+1 ))
echo -n " $commande :"
case $etat
in
D) echo "endormi  => ininterruptible  D" ;;
S) echo "endormi S" ;;
R) echo "en cours R" ;;
```

```
T) echo "stop T" ;;
Z) echo "zombi Z" ;;
*) echo "inconnu" ;;
esac
done
rm tempfile
fi
```

02/ Exemple :

L'objectif de cet exercice est de créer une commande ikill ne prenant pas d'argument mais demandant un nom de programme à l'utilisateur et tuant ce programme.

a/ en langage C

```
#include<signal.h>
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<sys/types.h>
#include<stdlib.h>
int main(){
system("ps -u");
kill(getpid(),SIGKILL);
}
int getpid(){
FILE *fp;
char name[300];
char temp[300];
char pid[300];
int p;
printf("donner le nom du processus   ");
scanf("%s",name);
strcpy(temp,"ps -Af | grep $USER | grep ");
strcat(temp,name);
strcat(temp," | grep -v grep | tr -s ' ' | cut -f 2 -d ' ' ");
fp=popen(temp,"r");
fgets(pid,8,fp);
p=atoi(pid);
return p;
pclose(fp);

}
```

b/ Sa traduction en script Shell

```
#!/bin/bash
ps u
echo -n "Quel processus voulez-vous supprimer: "
read name

PID=$(ps -Af | grep $USER | grep " $name" | grep -v grep | tr -s ' '
| cut -f 2 -d ' ')
if [  $(echo $PID | wc -c ) -eq 1 ]
then
	echo "pas de processus $name"
	exit 1
fi

echo $PID | sed 's/ /\n/g' > mytmp
n=$(cat mytmp | wc -l)
i=1
while [ $i -le $n ]
do
	currentPID=$(echo $PID | cut -f $i -d ' ' )
	if [ $currentPID -eq $$ ]
	then
		ps aux | grep "$currentPID" | grep -v grep
		exit 0
	fi
		ps aux | grep "$currentPID" | grep -v grep
		kill -9 $currentPID
	i=$(expr $i + 1)
done
```

03/ En vous inspirant sur les exemples précédents : Ecrire le script shell réveille capable de réveiller tous vos processus qui ont été suspendus par un signal SIGSTOP :

Version en Langage C :

```
#include<unistd.h>
#include<stdio.h>
#include<signal.h>
#include<sys/types.h>

int main(){
```

```
system ("ps -u");
int pid;

printf("Processus?      ");
scanf("%d",&pid) ;
kill (pid,SIGSTOP);
printf("Processus suspendu %d\n",pid);
system ("ps -u");
system("sleep 4");
kill (pid,SIGCONT);
printf("Poursuivre lexecution %d\n",pid);
system ("ps -u");

}
```

« Ceux qui ne comprennent pas Unix sont condamnés à le ré-inventer, lamentablement. » – Henry Spencer

XI/ Conclusion.

Les buts fixés pour l'utilisation de ce polycopié sont les suivants :

- Se familiariser avec ces systèmes d'exploitation et maitriser les commandes de bases.

- Ecrire des scripts (programmes) afin d'automatiser des tâches.

- Se familiariser avec ce système multitâche par l'intermédiaire de la gestion de processus grâce aux fonctions systèmes.

- Comprendre et maitriser la programmation de scripts (sh, ksh, bash, csh, tcsh).

- Un Bon travail c'est mettre en pratique tous ces conseils et astuces

XII/ Références

D.Benhaddouche Cours Système Unix Licence 2eme année Informatique Département Informatique USTO. 2010-2019.

D.Benhaddouche Travaux Pratiques Système d'-Exploitation I Licence 2eme année Informatique Département Informatique USTO. 2010-2019.

D.Benhaddouche Travaux Pratiques Système d'-Exploitation II Licence 3eme année Informatique Département Informatique USTO 2010-2019.

Canteaut(A.) Programmation en langage C - https://www.rocq.inria.fr

Moussel (P.). – Le langage C. – http://www.mrc.lu/LangC/.

François Morain « Principes de base des systèmes Unix » Enseignements à l'École polytechnique Paris.

Jean-Philippe Proux- Alain Perronnet- Cours Unix- Ecole Doctorale de Sciences Mathématiques de Paris Université Pierre et Marie Curie (Paris 6).