Chapitre 7 : Bash

Bash = shell = interface utilisateur du système
C'est aussi un langage de programmation

Steve Bourne (Bell Labs) : bash 1975

Bourne Again Shell

7.1 – Présentation

Bash permet d'écrire de petits programmes :

- faciliter des taches répétitives
- administration du système

Bash

- Bash est un shell :
  - Il gère la connexion avec le système (intégration dans la fenêtre XTerm ou PuTTY)
  - Il attend et exécute les commandes Unix
- C'est un langage de programmation :

  ses instructions = commandes Unix

  + structures de contrôle : tests, boucles...

  + variables, fonctions...

Emploi en tant que langage

- Exactement comme en C :
  - On inscrit des instructions dans un programme
  - Quand on lance ce programme, ça exécute les instructions
- Donc l'exécution d'un programme bash lance des commandes Unix
- La différence : pas de compilation

Applications de bash

- Bash sert essentiellement à écrire des programmes relatifs aux fichiers, dossiers, comptes, logiciels, services, etc. :
  - Tout ce qui concerne l'administration du système
  - Tout ce qui serait un peu lourd à faire en C

Unix et les scripts

- Une partie du système Unix est programmée sous forme de scripts (qui lancent des binaires compilés) :
  - le démarrage et l'arrêt des services : `/etc/init.d/*`
  - le réseau : `/etc/network/if-up.d/*`
  - les tâches planifiées : `/etc/cron.daily/*`
  - certaines commandes et logiciels :

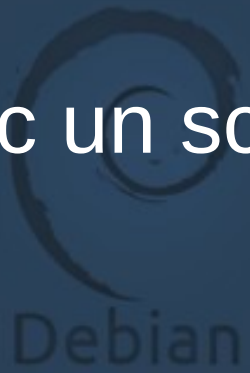
    ```
    file /usr/bin/* | fgrep 'shell script'
    ```
- parce qu'un script est 1000 fois plus simple à écrire qu'un programme compilé... quand c'est possible

Scripts

- Un programme bash est appelé script
  - On peut les suffixer par .sh, exemple sauver.sh
- Un script est un fichier de type texte :
  - Édité avec emacs, vi, geany, nano, gedit...
  - Commençant obligatoirement par `#!/bin/bash`
  - Rendu exécutable par : `chmod u+x prog.sh`

7.2 – Un script d'exemple

Illustration avec un script tout simple



Un premier exemple

- Taper ceci dans un fichier appelé essai1.sh

```
#!/bin/bash
# mon premier script bash

echo bonjour tout le monde
```

- Taper `chmod u+x essai1.sh`
- Taper `essai1.sh` (ou `./essai1.sh`)

À quoi sert le chmod u+x ?

- Ce n'est pas une compilation
- `chmod u+x` rajoute seulement l'autorisation d'exécuter le script : u=vous + x=exécuter
  - Par défaut, un fichier texte n'est pas exécutable, il faut accorder ce droit pour qu'il devienne un script
  - Ce droit reste en place même après édition : le chmod n'est à faire qu'une seule fois
  - NB : on parlera des droits sur les fichiers en période P3

À quoi sert le `#!/bin/bash` ?

- C'est un identifiant, un code (magic number) qui indique la nature du fichier (voir TP1)
  - Ex : un fichier gif commence par `gif89a`
  - Ex : un fichier compilé commence par `.ELF`
- La commande `file` se base sur le début des fichiers pour afficher leur nature, exemple :

```
file essai1.sh
essai1.sh : POSIX shell script, ASCII text executable
```

./essai1.sh ou essai1.sh ?

- Pourquoi faut-il taper `./prog` pour lancer un programme ?
  - Les commandes Unix sont dans différents répertoires : /bin, /usr/bin, /usr/local/bin, /sbin, etc.
  - Quand on tape le nom d'une commande, bash la cherche en tant que programme dans tous ces répertoires

```
prompt$ jezfooizejf
bash: jezfooizejf : commande introuvable
prompt$ ls    ==> lancement de /bin/ls
```

./prog ? suite

- Vos programmes et scripts ne sont pas dans ces dossiers, donc bash ne les trouve pas
  - Soit indiquer leur nom complet au lancement :
    - Taper `./monprog`
  - Soit ajouter `.` (répertoire courant) à la liste des dossiers de recherche des commandes
- Si un programme a été déplacé, taper `hash -r`
  - Ça fait réapprendre l'emplacement des programmes

Ajouter . à la liste des dossiers

- La liste des dossiers contenant les commandes est dans la variable bash appelée PATH
  - `echo $PATH` affiche /usr/local/bin:/usr/bin:/bin: …
  - Les chemins sont collés, séparés par :
- Il va falloir modifier cette variable
  - Éditer le fichier `~/.bashrc`
  - Y rajouter la ligne `export PATH=.:$PATH`
  - Vérifier que c'est ok en tapant : `source ~/.bashrc`

    Ne surtout pas se déconnecter s'il y a une erreur

Qu'est-ce que ~/.bashrc ?

- C'est un fichier de configuration de votre compte
  - Il est exécuté quand vous vous connectez
- Il donne les options de bash, par exemple :
  - Variable PATH : emplacement des commandes
  - Variables PS1 et PS2 : les messages d'invite
  - Alias des commandes : alias egrep='egrep …'
  - De nombreuses autres choses liées au système

7.3 – Les variables bash

Une variable permet de stocker une chaîne :
nom de fichier, message, nombre...

Les variables en bash

- On vient de voir la variable PATH, on peut en définir d'autres soi-même :

```
#!/bin/bash
# affectation d'une variable
Message='Bonjour tout le monde'

# utilisation d'une variable
echo $Message
```

Affectation d'une variable

- NOMVAR=VALEUR
  - ne mettre aucun espace avant et après le =
- Il y a différentes possibilités pour la valeur :
  - Un seul mot
    - Ex : `reponse1=oui`
  - Une chaîne entre apostrophes (*simple quotes*)
    - Ex : `reponse2='ouais, c'\''est pas faux'`
  - Une chaîne entre guillements (*double quotes*)
    - Ex : `reponse3="il a dit $reponse2"`

Différence entre '...' et "..."

- Les chaînes "..." peuvent contenir des variables $NOM qui seront remplacées par leurs valeurs

```
nom=nerzic
message="Bonjour Mr $nom"
```

- Les chaînes '...' ne modifient rien, c'est mis tel quel dans la variable

```
prix='ce smartphone vaut 3$CAN'
```

Valeur d'une variable

- Pour obtenir le texte contenu dans une variable, il suffit de mettre un $ devant son nom

```
message='Salut tout le monde'
echo "le message est : $message"
```

- Dans certains cas (variable suivie d'un texte), il faut encadrer le nom de la variable par { }

```
echo "le message est XXX${message}XXX"
```

Pourquoi des variables ?

- Pour pouvoir moduler un traitement ex : MkProjetC

```
#!/bin/bash
# ce script crée un projet C : dossier et source
projet=test1

mkdir $projet
cd $projet
echo '#include <stdlib.h>'       >  ${projet}.c
echo 'int main() {'              >> ${projet}.c
echo '   return EXIT_SUCCESS;'   >> ${projet}.c
echo '}'                         >> ${projet}.c
```

Lancement du script

- On lance le script en tapant son nom

  Ex : MkProjetC

  => le script exécute les commandes suivantes :

```
projet=test1
mkdir test1
cd test1
echo '#include <stdlib.h>' >  test1.c
echo 'int main() {'        >> test1.c
...
```

Entrées et sorties d'un script

- Sortie = affichage sur écran (redirigeable) :
  - `echo message`
  - `echo -n message` affiche sans revenir à la ligne
- Entrée = lecture clavier (redirigeable) :
  - `read mot1 mot2 … reste`

    Affecte la variable mot1 avec le 1er mot de la ligne, mot2 avec le suivant et reste avec ce qui reste
  - `read -p "`*`message`*`" `*`variables`*`...` affiche *message* en tant que prompt

Le script devient

- On demande de taper le nom du projet :

```
#!/bin/bash
# ce script crée un projet C : dossier et source
read -p "Nom du projet : " projet

mkdir $projet
cd $projet
echo '#include <stdlib.h>'       >  ${projet}.c
echo 'int main() {'              >> ${projet}.c
echo '    return EXIT_SUCCESS;'  >> ${projet}.c
echo '}'                         >> ${projet}.c
```

7.4 – Paramètres d'un script

Un script peut recevoir des paramètres

Paramètres de lancement

- Au lieu d'une saisie clavier, on lui passe un paramètre au lancement :

```
#!/bin/bash
# ce script crée un projet C : dossier et source
projet=$1

mkdir ${projet}
cd ${projet}
echo '#include <stdlib.h>'       >  ${projet}.c
echo 'int main() {'              >> ${projet}.c
echo '   return EXIT_SUCCESS;'   >> ${projet}.c
echo '}'                         >> ${projet}.c
```

Passage de paramètre

- On lance le script ainsi :

  NomDuScript Parametre

  - Ex : `MkProjetC CalculTVA`

  => `$1` vaut ce paramètre et le script exécute ceci

```
projet=CalculTVA
mkdir CalculTVA
cd CalculTVA
echo '#include <stdlib.h>' >  CalculTVA.c
...
```

Moins modulaire mais plus court

- On peut mettre $1 partout :

```
#!/bin/bash
# ce script crée un projet C : dossier et source

mkdir $1
cd $1
echo '#include <stdlib.h>'       >  $1.c
echo 'int main() {'              >> $1.c
echo '   return EXIT_SUCCESS;'   >> $1.c
echo '}'                         >> $1.c
```

Tous les paramètres sont gérés

- La variable spéciale $1 reçoit le 1er paramètre
  - Ex : MkProjetC toto => $1 = toto
- La variable spéciale $2 reçoit le 2e paramètre
  - Ex : MkProjetC toto titi tutu => $2 = titi
- etc... $9 reçoit le 9e paramètre
- S'il y en a d'autres, alors on emploie shift
  - shift décale $2 dans $1, $3 dans $2... $n+1 dans $n

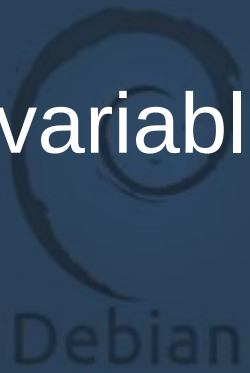
Paramètres vides

- Il faut se méfier des paramètres et des variables : s'ils sont vides ou s'ils contiennent plusieurs mots. Exemple :
  - `projet='Far Cry 5'`
  - `mkdir $projet`

    => ne crée pas un dossier appelé « Far Cry 5 », mais trois dossiers : l'un appelé Far, l'autre Cry et le 3e appelé 5 car dans mkdir, les mots se séparent !
- Solution : encadrer chaque variable par "…"
  - `mkdir "$projet"`

7.5 – Tests sur les variables

Réagir si une variable est vide, etc.



Rajouter des tests

- Si l'utilisateur oublie de fournir un paramètre ?
  - $1 est vide => le script plante ou fait n'importe quoi
  - Ex : mkdir échoue, echo crée un fichier appelé .c
- Rajouter un test avant le « cœur » du script :

```
if test "$1" = ""       # remarquer les "..."
then
   echo "erreur : donnez le nom du projet"
   exit 1
fi
```

If test ; then ; fi

- La syntaxe est très différente du langage C
  - On verra l'intérêt dans un prochain cours
- Tests possibles sur des chaînes :
  - chaine1 = chaine2 attention, pas double ==
  - chaine1 != chaine2

  Piège : il faut un espace de chaque côté du = ou !=
    - `if test $reponse=oui` est toujours vrai !!
    - `if test $reponse = oui` fait le test correctement

Exit ou else ?

- L'instruction exit *nb* fait quitter immédiatement
  - *nb* est un code d'erreur : 0 = EXIT_SUCCESS, 1 = EXIT_FAILURE
- Donc on a le choix entre ces deux schémas

```
if test "$1" = ""
then
   echo "message"
   exit 1
fi
suite...
```

```
if test "$1" = ""
then
   echo "message"
else
   suite...
fi
```

Choix

- On préférera le schéma de gauche (avec exit, sans else) pour tester les paramètres avant de lancer les traitements du script
  - Raison : le programme est plus lisible : tous les tests de validité des paramètres en séquence puis le cœur du script
  - Sinon : imbrication des else, if qui peut rendre le programme illisible

Tests sur les fichiers et dossiers

- Si le paramètre désigne un dossier ou fichier existant ?

```
if test -d "$1"          # $1 est-il un dossier ?
then
   echo "erreur : $1 existe déjà"
   exit 1
fi
if test -f "$1/$1.c"   # est-ce un fichier ?
then
   echo "erreur : $1/$1.c existe déjà"
   exit 1
fi
Suite...
```

7.6 – Lancement de commandes

Comment échanger des données avec des commandes ?

Commandes et redirections

- Bash sert à lancer des commandes :
  - Ex : `mkdir "$1"`
- Redirection d'entrée
  - Ex : `wc -l < "$1"`
  - Ex : `echo "$1" | wc -l`

    Affiche le nombre de lignes contenues dans $1
- Redirection de sortie
  - Ex : `ls *.c > listing`

Redirection vers une variable

- On peut rediriger la sortie vers une variable
  - Ex : compter les lignes du fichier indiqué par $1

```
wc -l "$1"                  => résultat à l'écran
NbL=$(wc -l "$1")           => résultat dans NbL
```

- La syntaxe `NOMVAR=$(CMDE)` redirige la sortie de la commande vers la variable
- Variante ancienne : `NOMVAR=`CMDE``

Quelques applications

- Nom du plus gros fichier :

```
nom=$(ls -l | awk '{print $5,$9}' |
sort -n | tail -n 1 | cut -d' ' -f2)
```

- Création d'un dossier en fonction de la date

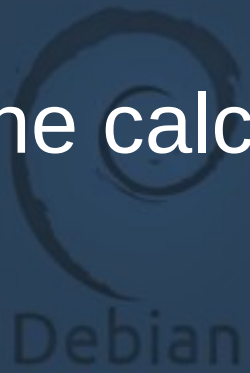
```
mkdir $(date +%Y-%m-%d)
```

- Lancement de l'éditeur geany sur tous les sources C que vous avez :

```
geany $(find ~ -name '*.c')
```

7.7 – Calculs arithmétiques

Bash possède une calculatrice minimale



Calculs arithmétiques

- La syntaxe $((expression numérique)) permet d'effectuer de petits calculs

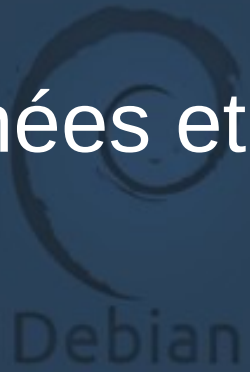
```
NbProcs=$(ps -def | wc -l)
NbProcs=$((NbProcs – 1))
```

!! Ne pas confondre $(cmde) avec $((calcul))

- Les opérateurs sont seulement + - * / et % et ne travaillent que sur des entiers

7.8 – Itérations

Boucles bornées et non-bornées



Boucles non bornées

- La structure while test ; do... ; done ressemble à celle du langage C :

```
read -p "On continue ? (O/N)" reponse
while test "$reponse" = "O"
do
   echo 'Ok, alors on continue...'
   ...
   read -p "Et là, on continue ? (O/N)" reponse
done
echo 'Et voilà, c'\''est fini'
```

Boucles comme en C

- Il existe aussi les boucles du langage C :

  for ((init ; condition de continuation ; avancement))

  do

  Action...

  done

- Exemple :

```
for ((x=0 ; x<9 ; x++))
do
   echo x = $x
done
```

Boucles d'énumération

- Il existe une boucle for très utile mais assez différente de celles du C :
  - Le principe est de parcourir une énumération, élément par élément
  - Pour chaque élément x de la liste, faire ceci sur x

```
for x in liste...
do
   Traitement sur $x
done
```

```
for mot in non mais quoi
do
   echo -n "$mot allo "
done
```

Utilité de cette boucle

- On peut traiter tous les paramètres

```
for param in "$@" ; do ..."$param"... ; done
```

"$@" est la liste de tous les paramètres : $1 $2...

- On peut traiter tous les fichiers

```
for fichier in * ; do ..."$fichier"... ; done
```

* (joker bash) est la liste de tous les fichiers

- On peut traiter le résultat d'une commande

```
for mot in $(cmde) ; do ..."$mot"... ; done
```

$(cmde) est la liste des mots écrits par la commande