Programmation Unix 1 – cours n°1

Edouard THIEL

Faculté des Sciences

Université d'Aix-Marseille (AMU)

Septembre 2016

Les transparents de ce cours sont téléchargeables ici :
http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/unix/

Lien court : http://j.mp/progunix

Présentation de l'UE

- 5 séances C/TD/TP
- Pas de contrôle continu
- Examen final : droit aux notes de CM/TD/TP, pas aux annales ni aux livres
- Contenu :
  - Langage de script bash
  - Utilisation, organisation système Unix
- Objectifs :
  - Maîtriser bash
- Sur la page web du cours : transparents, annales corrigées, liens, etc.
- Prise de notes : noter ce qui semble important, renvoyer au numéro de transparent quand beaucoup d'informations.

Plan du cours n°1

1. Système d'exploitation
2. Le shell
3. Commandes Unix
4. Redirections et tubes
5. Les scripts
6. Éléments du langage bash

1 - Système d'exploitation

Un système d'exploitation est un ensemble de logiciels qui

- prend le contrôle de la machine après le boot ;
- partage et gère les ressources de la machine pour les autres logiciels : processeur, mémoire, disque, réseau, écran, etc ;
- rend transparent les entrées-sorties pour les autres logiciels.

Unix désigne une famille de SE. Quelques propriétés :

- Multi-utilisateur, phase d'identification, droits.
- Manipulation du système via un interpréteur de commande : un "shell".
- Unité élémentaire de gestion des ressources = le fichier.
- Unité élémentaire de gestion des traitements = le processus.

Historique Unix

1964 Projet "Multics" de S.E. multi-tâche (MIT, General Electric, Bell Labs d'AT&T) — terriblement complexe à utiliser

1969 Ken Thompson et Denis Ritchie (Bell Labs) écrivent 1 S.E. multi-tâche sur un ordi. de récupération (DEC PDP 7 de 1964)

Brian Kernigham (Bell Labs) l'appelle "Unix" par opposition à Multics ; Multics abandonné

1970-71 V1 : Réécriture sur PDP11 (16 bits, 24K Ram, 512K DD)

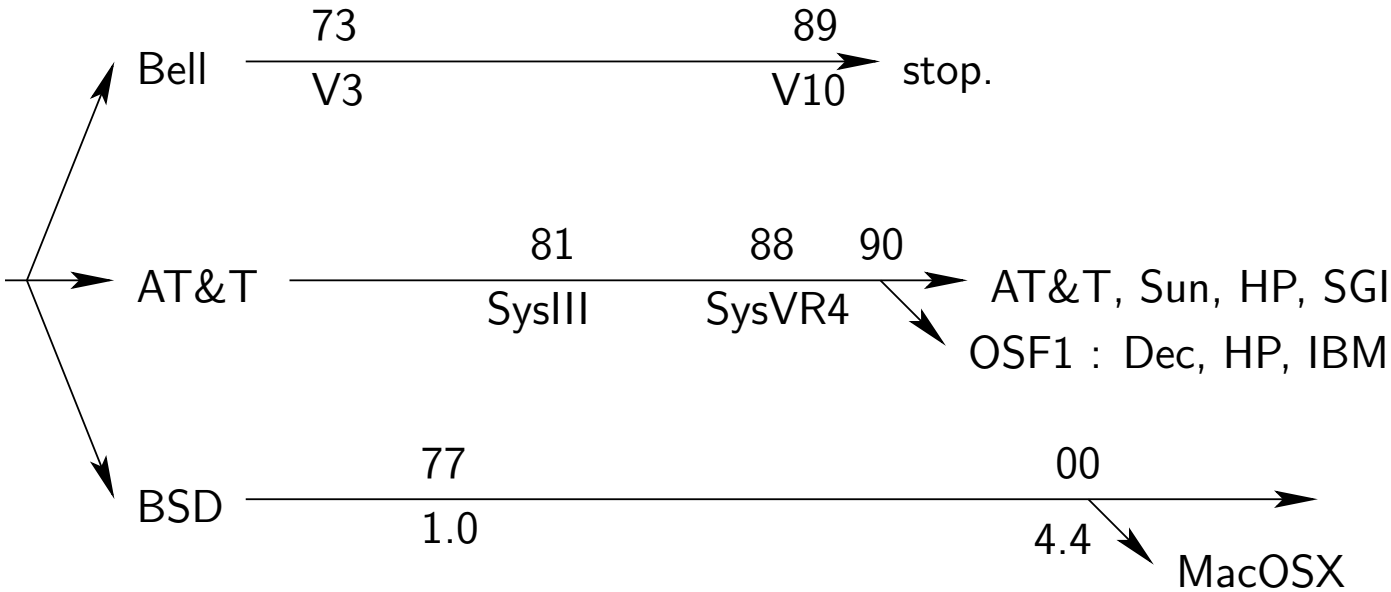
Ken Thomson crée langage B (inspiré de BCPL) ; réécrit Unix

1972-73 Denis Ritchie crée langage C (inspiré de B)

V2 : Ken Thomson réécrit entièrement Unix en C

Les sources sont fournis à : Bell ; AT& T ; Univ. Californie à Berkeley $\rightarrow$ 3 branches principales de développement.

Branches principales de développement Unix



Historique complet et documents : www.levenez.com

Normalisation

POSIX (IEEE, depuis 1988)

- 3 normes, 17 documents
- API ; commandes et utilitaires Unix ; tests de conformité
- documents et certifications coûteux

Logiciels libres

Systèmes propriétaires, et souvent très chers

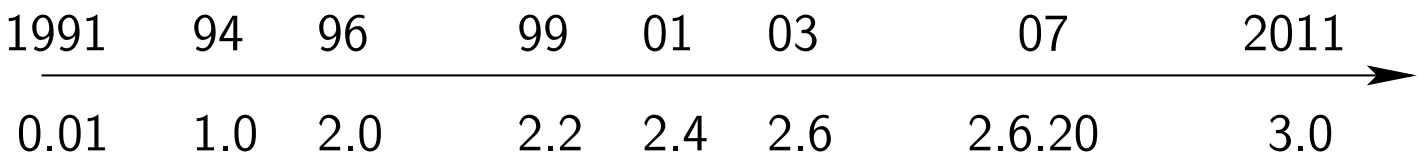
→ Émergence Logiciels Libres : Richard Stallman

- FSF (Free Software Foundation),
- Projet GNU (Gnu is Not Unix) = réécrire Unix
- Licences GPL, CC, etc

GNU/Linux

Linux : Linus Torvalds

Étudiant Finlandais, étudie système MINIX (auteur: Tanenbaum) écrit 1 noyau pour projet Master 1, puis appel à contribution sur internet.



Distributions = noyau Linux + utilitaires GNU + logiciels libres + documentation + installeur + utilitaires :

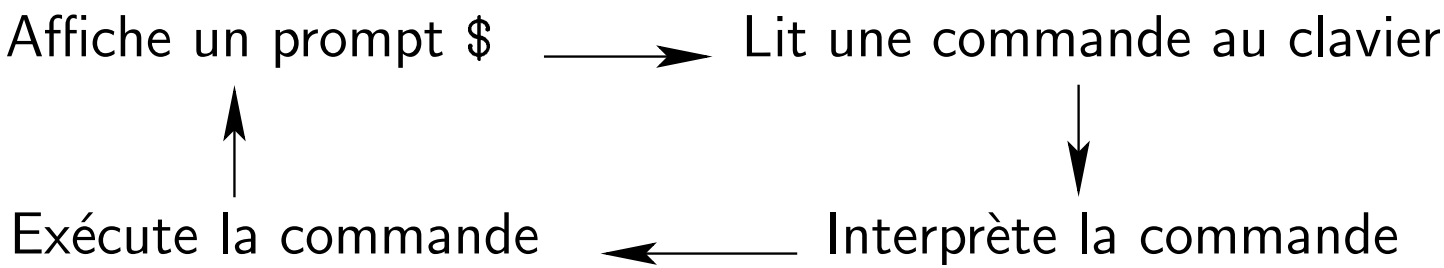
Ubuntu, Debian, Redhat, Fedora, Suse, Slackware, etc

99% Top 500 supercalculateurs, serveurs, Android, liseuses, box internet, TV, voitures, etc

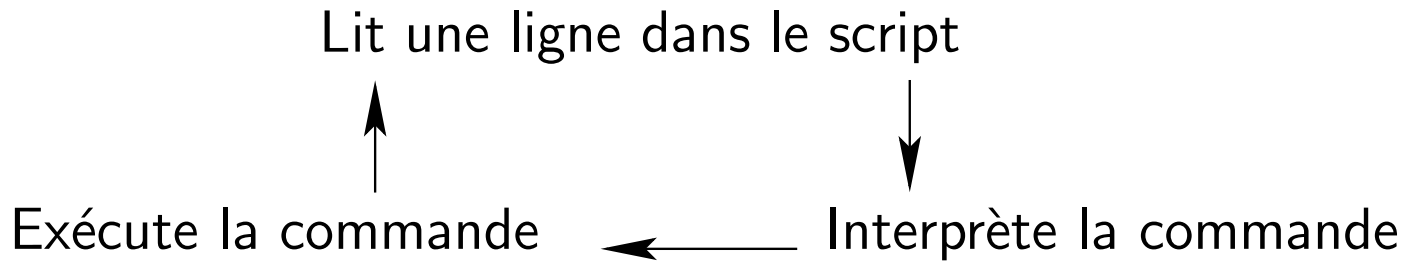
2 - Le shell

L'interpréteur de commande est très important sous Unix.
Il permet de manipuler des fichiers et des processus avec des commandes.

- Mode interactif : dans un terminal



- Mode script (fichier de commandes) :



Shells Unix et langages de script

- Le "Bourne-shell" : langage sh
  - créé par Steeve Bourne en 1977,
  - le plus ancien, le shell par défaut
  - syntaxe puissante mais rugueuse
- Le "C-shell" : langage csh
  - syntaxe se rapprochant du C
  - pas de fonctions ; moins expressif
  - Document pourquoi préférer sh à csh
- Le "Korn-shell" : langage ksh
  - sur-ensemble de sh, plus récent
- etc
- Clones libres :
  - csh $\rightarrow$ tcsh : turbo-csh
  - sh $\rightarrow$ bash : Bourne-again shell

Bash

- 1988 Brian Fox, pour le projet GNU
- devenu le shell de référence, le plus utilisé (Unix libres, propriétaires, MacOSX, Cygwin).
- sur-ensemble de sh, incluant des possibilités de ksh et csh

Langage bash : puissant et expressif

- interprété, basé sur les substitutions
- fonctions, récursivité
- types simples : chaînes, entiers, tableaux
- contrôle de flux : redirections, tubes
- exécution séquentielle ou parallèle
- permet d'augmenter le système.

Syntaxe bash

- Découpage sur les "blancs" (espace, tab, rc)
- Caractères réservés : ' " ` () {} [] \ & ~ | $ # ; ...
- Langage orienté commandes : la syntaxe privilégie la gestion des commandes sur les autres aspects
- En C, on appelle des instructions et on évalue des expressions ; en bash, on exécute des commandes et on regarde si elles ont réussi.

3 - Commandes Unix

Certaines commandes sont intégrées à bash : les "builtins"

`echo "message"`	affiche le message à l'écran
`read toto`	lit une ligne au clavier, puis la mémorise dans la variable `toto`

Exemple :

```
$ echo "Quel est votre nom ?"
Quel est votre nom ?
$ read nom
Thiel
$ echo "Votre nom est $nom"
Votre nom est Thiel
```

Commandes Unix de base (1)

La plupart des commandes sont des programmes externes : au moins 800 (dans `/bin`, `/usr/bin`, etc)

`cat fichier`	affiche le fichier
`more fichier`	idem + pause après chaque écran
`ls`	liste le répertoire courant
`ls rep`	liste le répertoire `rep`
`cd rep`	change le répertoire courant
`cd ..`	va dans le répertoire parent
`pwd`	affiche le répertoire courant
`mkdir rep1`	crée le sous-répertoire `rep1`
`rmdir rep1`	détruit le sous-répertoire `rep1` si vide

Commandes Unix de base (2)

`cp f1 f2`	copie fichier `f1` en fichier `f2`
`cp f1 f2 ...fn rep`	copie les fichiers dans `rep`
`mv f1 f2`	renomme fichier `f1` en fichier `f2`
`mv f1 f2 ...fn rep`	déplace les fichiers dans `rep`
`rm f1 ...fn`	détruit les fichiers
`vi f1`	ouvre `f1` avec éditeur `vi`
`touch f1`	change date de modification de `f1` ou crée un fichier vide
`chmod droits f1 ...fn`	change les `droits` des fichiers
`chown propri f1 ...fn`	change le `propriétaire` des fichiers
`chgrp groupe f1 ...fn`	change le `groupe` des fichiers

Documentation

La plupart des commandes sont documentées.

- Pour les commandes builtin : `help`

  ```
  help echo
  ```

- Pour les commandes externes : `man` ou `info`

  ```
  man ls
  info gcc
  ```

- Les commandes d'aide sont elles-mêmes auto-documentées :

  ```
  help help
  man man
  info info
  ```

Arguments

- La plupart des commandes ont des arguments :

  `ls -l -a rep` affiche infos sur chaque fichier de `rep`

  `cp -f f1 f2` force copie si `f2` existe déjà

- Dans certaines commandes les options peuvent être regroupées :

  ```
  ls -la rep
  ```

- Dans certaines commandes les options existent au format long ou court :

  ```
  ls --recursive --all
  ls -R -a
  ls -Ra
  ```

- Dans la documentation : indication des options entre `[ ]`

  ```
  ls [option] ... [file] ...
  cp [option] ... source ... directory
  ```

4 - Redirections et tubes

• Le shell permet de rediriger facilement l'entrée standard depuis 1 fichier (sans que le programme appelé ne le sache)

```
commande arguments < fichier
```

→ Lecture depuis fichier, écriture à l'écran

• De même pour la sortie standard :

```
commande arguments > fichier
```

→ Lecture au clavier, écriture dans fichier

• On peut combiner :

```
commande arguments < f1 > f2
```

Variantes

`>`		crée le fichier si n'existe pas ; sinon écrase ou échoue, selon la configuration (`set -o noclobber`)
`>|`		crée ou écrase (force)
`>>`		ajoute à la fin d'un fichier existant ou échoue (append)
`0<`	`1>`	synonymes de < et >
`2>`		redirige la sortie d'erreurs
`<<`	`<<<`	"here document", vus plus loin

Tubes

• Les tubes sont des canaux de communications très efficaces fournis par le système.

Ils permettent de relier la sortie d'une commande sur l'entrée d'une autre commande, via une mémoire tampon gérée par le système.

```
com1 arguments | com2 arguments
```

Ceci revient à faire (en nettement moins efficace) :

```
com1 arguments > tmp &
com2 arguments < tmp
rm -f tmp
```

& = en tâche de fond
com1 et com2 sont donc exécutés en parallèle

• On peut combiner :

```
ls -l | sort -r | head -5
```

```
cat < f1 | sort -r | head -20 > f2
```

Filtres

Certaines commandes sont des *filtres* :

- elles lisent sur l'**entrée standard** (le clavier par défaut)
- elles affichent sur la **sortie standard** (l'écran par défaut)

`cat`	(sans nom de fichier)
`..`	recopie chaque ligne tapée après RC
`..`	
`^D`	fin de fichier

Autres exemples de filtres : `sort`, `uniq`, `tr`, `head`, `tail`, `cut`, `paste`, `join`

$\rightarrow$ Les filtres sont spécialement conçus pour être reliés par des tubes

5 - Les scripts

Script = fichier de commandes, exécuté par un shell

Il faut dire au système quel shell utiliser :
première ligne avec mot magique `#!` "shebang"

```
#! /chemin/interpréteur
```

Exemple : `/bin/bash` ou `/bin/csh` ou `/usr/bin/python`

Le script doit être rendu exécutable : `chmod +x fichier`

Un script est une commande :
en écrivant des scripts on "augmente le système".

Script bash

Exemple : script `hello.sh`

```
#! /bin/bash
# Mon premier script
echo "Hello world!"
exit 0   # Le script a réussi
```

- Exécution :

```
$ chmod +x hello.sh     # une seule fois
$ ./hello.sh
Hello world!
```

Nature d'un fichier

- Commande `file` : reconnaît la nature d'un fichier d'après son contenu ; mots magiques recensés dans `/etc/magic`

```
$ file hello.sh
hello.sh: Bourne-Again shell script, text executable
```

- Chercher les scripts dans le répertoire courant :

```
$ file * | grep script
```

Code de terminaison

- Lorsqu'une commande ou un script se termine, elle donne un code de terminaison :

  `exit [n]` en C : `exit (n);`

Signification :	0	succès
	1...255	échec

- On peut récupérer ce code avec `$?` :

```
$ ./hello.sh
Hello world!
$ echo $?
0
$ false
$ echo $?
1
$ true
$ echo $?
0
```

6 - Éléments du langage bash

- Branchement `if`
- Boucle `while`
- Commande `test`
- Variables

Le branchement `if`

```
if commande arguments
then
    echo "succès"
else
    echo "échec"
fi
```

Il faut un `;` ou un RC avant `then`, `else` et `fi`

→ La commande est exécutée. À sa terminaison, branchement selon `$?`.

Variantes :

```
if .. ; then .. ; fi
if .. ; then .. ; else .. ; fi
if .. ; then .. ; elif .. ; then .. ; else .. ; fi
```

La boucle `while`

```
while commande arguments
do
    echo "une itération"
done
```

Il faut un `;` ou un RC avant `do` et `done`

$\rightarrow$ La commande est exécutée. À sa terminaison, si `$?` est 0 (succès), le bloc `do .. done` est exécuté, puis itération.

Commande `test`

- Existe en builtin et en commande externe.
- Nombreuses options pour tester : fichiers, chaînes, entiers
- Évalue une expression par arguments, puis réussie ou échoue

→ appelé par `if` ou `while`

```
if test -f "hello.sh" ; then
    echo "Le fichier existe"
fi
```

- Variante : `[` est un lien sur `/bin/test`

```
if␣[␣-f␣"hello.sh"␣];␣then
```

→ attention aux espaces !

Inversion de résultat

On peut inverser le résultat d'une commande avec ! :

```
! commande arguments
```

Exemple :

```
$ false ; echo $?
1
$ ! false ; echo $?
0
```

Utilisation classique :

```
if ! commande arguments ; then ... ; fi
if ! test ... ; then ... ; fi
if ! [ ... ]; then ... ; fi
while ! commande arguments ; do ... ; done
```

Variables

Variables de type chaîne de caractères par défaut ; gestion mémoire automatique.

Créer une variable vide :

```
nom_var=
```

Créer une variable et lui affecter une valeur :

```
nom_var=valeur
```

accolé, pas d'espace autour du =

Si la valeur comporte des blancs, protéger avec "" ou ''

```
message="Bonjour les amis"
```

Lire une ligne au clavier et la stocker dans une variable

```
read nom_var
```

Substitution

Accéder à la valeur d'une variable : `$nom_var` ou `${nom_var}`

→ le shell substitue `$nom_var` par sa valeur

```
echo "Valeur de message : $message"
```

↓ substitution

```
echo "Valeur de message : Bonjour les amis"
```

↓ affichage

```
Valeur de message : Bonjour les amis
```

Protections

- Que se passe-t'il si on remplace "" par '' ?

→ La substitution n'est pas faite

- Afficher " → \"
  \ → \\
  $ → \$

```
echo "Valeur de message : \"$message\""
                    ↓ substitution
echo "Valeur de message : \"Bonjour les amis\""
                    ↓ affichage
Valeur de message : "Bonjour les amis"
```

Concaténation

On peut concaténer par substitution :

```
numero=17
rue="rue du midi"
adresse="$numero $rue"
```

On peut concaténer à la fin :

```
chemin="/usr"
chemin+="/bin"
```

Instrospection

Demander à bash la valeur d'une variable :

```
declare -p nom_var
```

Exemple :

```
$ foo="*"
$ declare -p foo
declare -- foo="*"
```

Détruire une variable :

```
unset nom_var
```