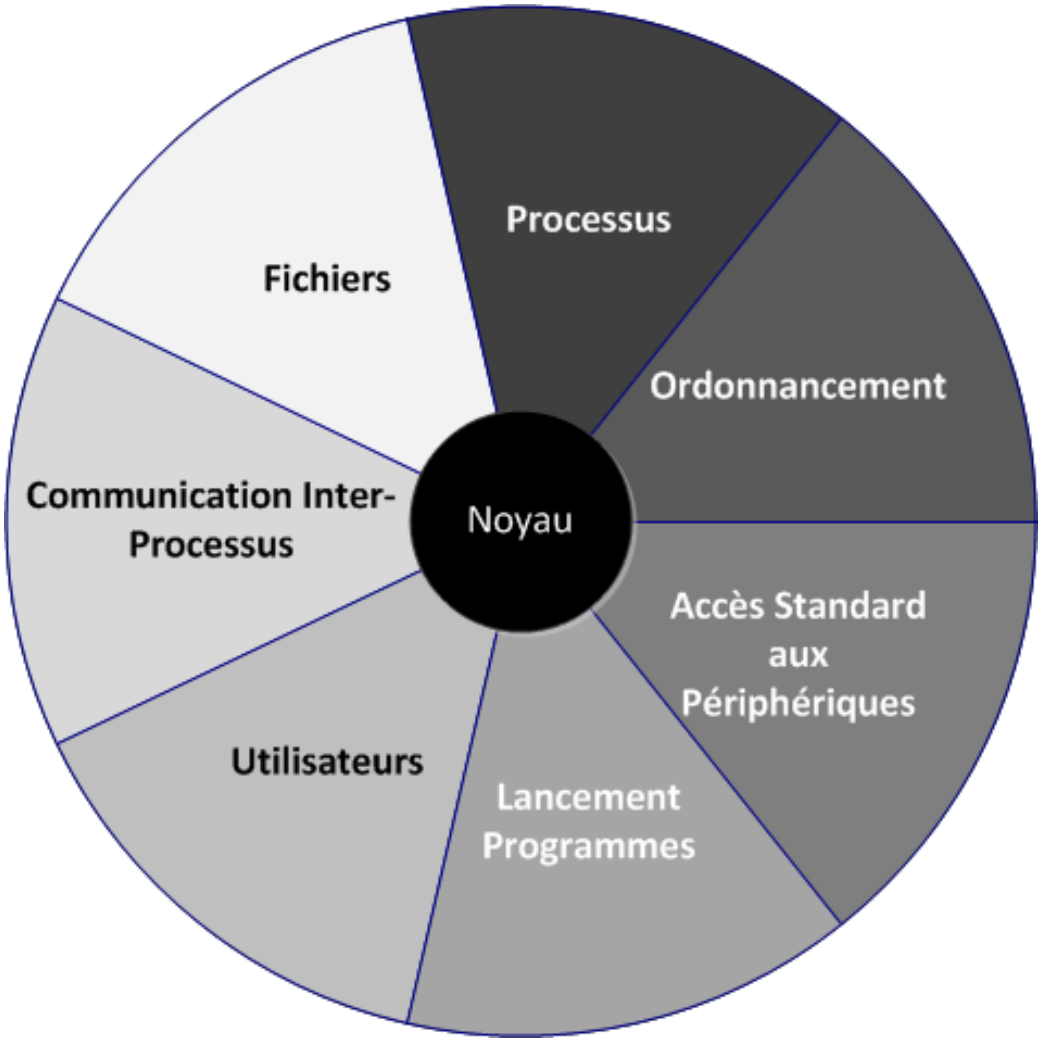
Architecture des Ordinateurs et Systèmes d'Exploitation

Partie 3 : Shell
Cours (suite) & TD

Fabrice BOISSIER & Elena KUSHNAREVA
2017/2018

fabrice.boissier@gmail.com
elena.kushnareva@malix.univ-paris1.fr

Les Systèmes d'Exploitation

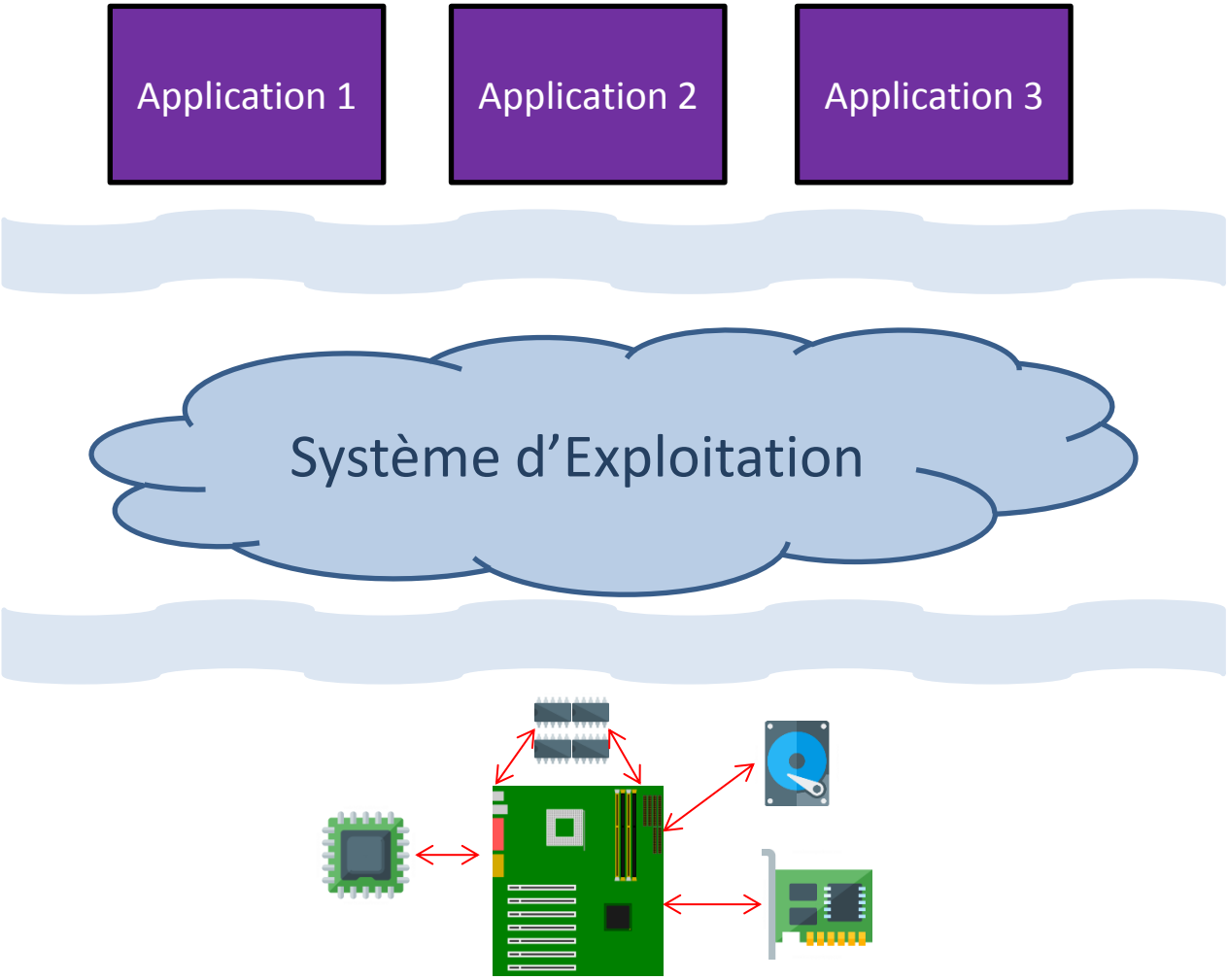


Les Systèmes d'Exploitation

Le Système d'Exploitation gère tout.

C'est un « système » qui « exploite » la machine/plateforme pour rendre des services aux utilisateurs.

Les Systèmes d'Exploitation



Les Systèmes d'Exploitation

- Fonctionnellement :

  Fourni des services utiles aux utilisateurs

- Techniquement :

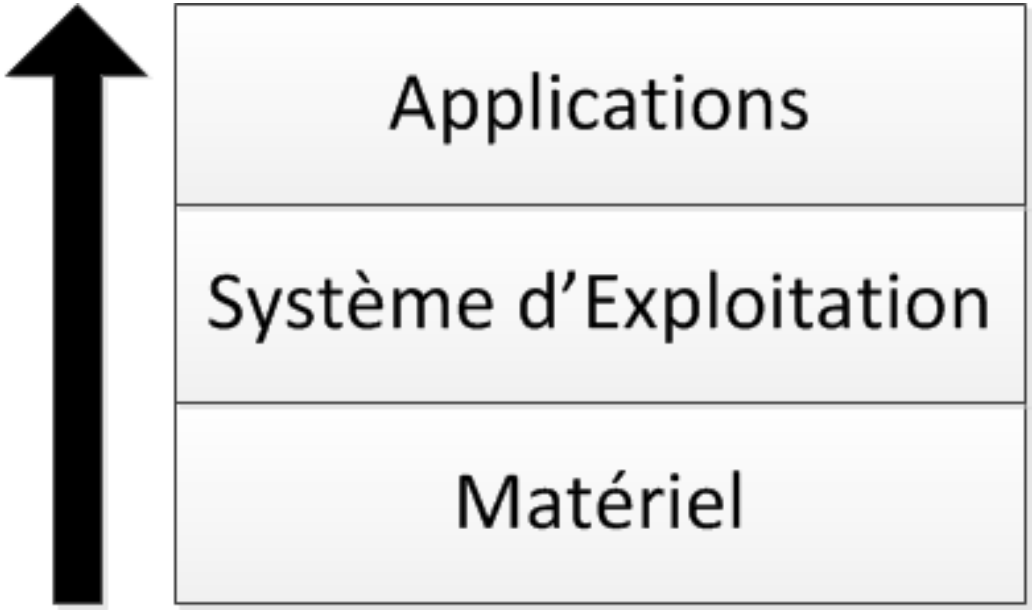
  Utilise le plus efficacement la plateforme

En offrant une surcouche indépendante du matériel

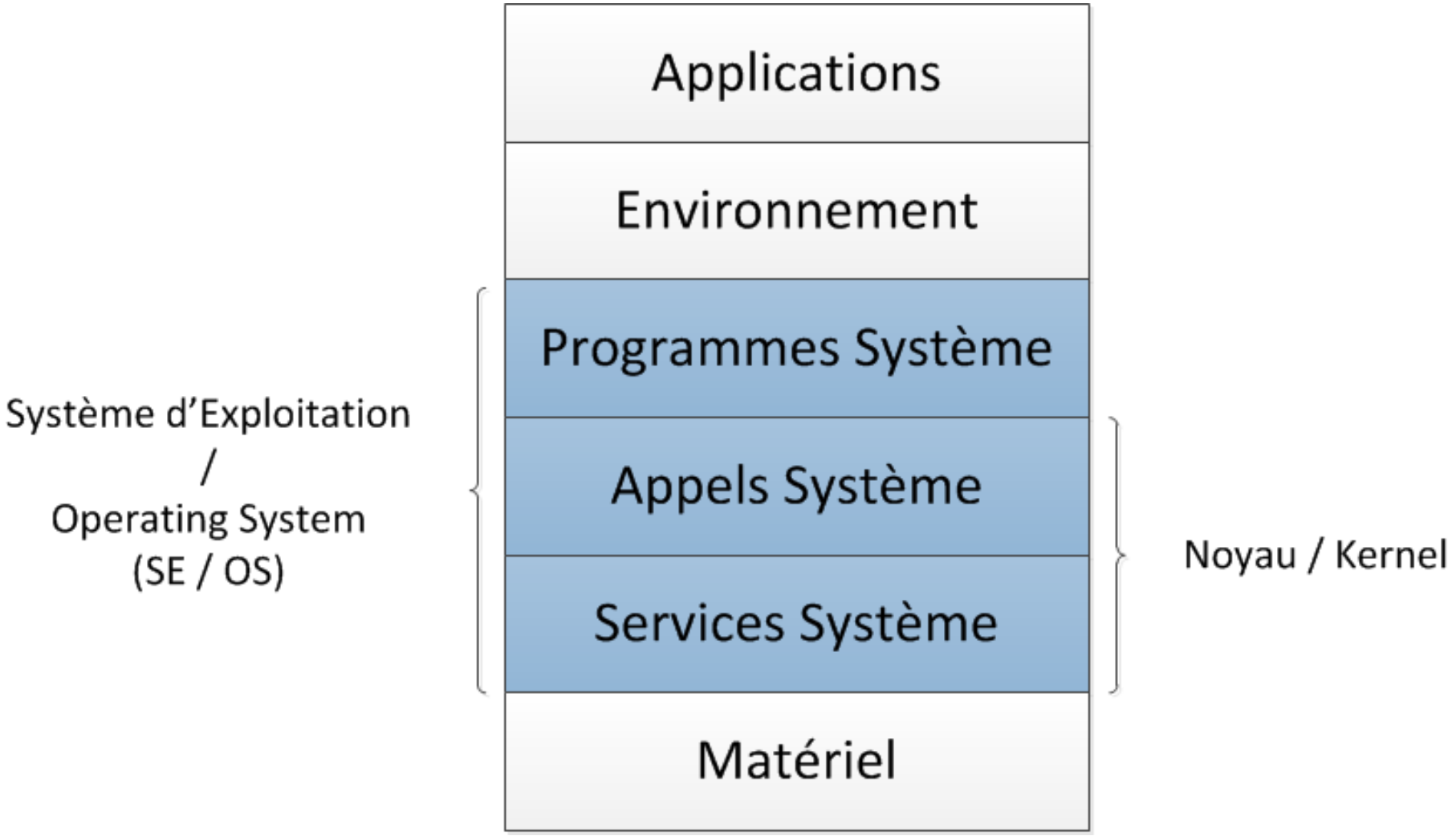
Les Systèmes d'Exploitation

Applications Portables

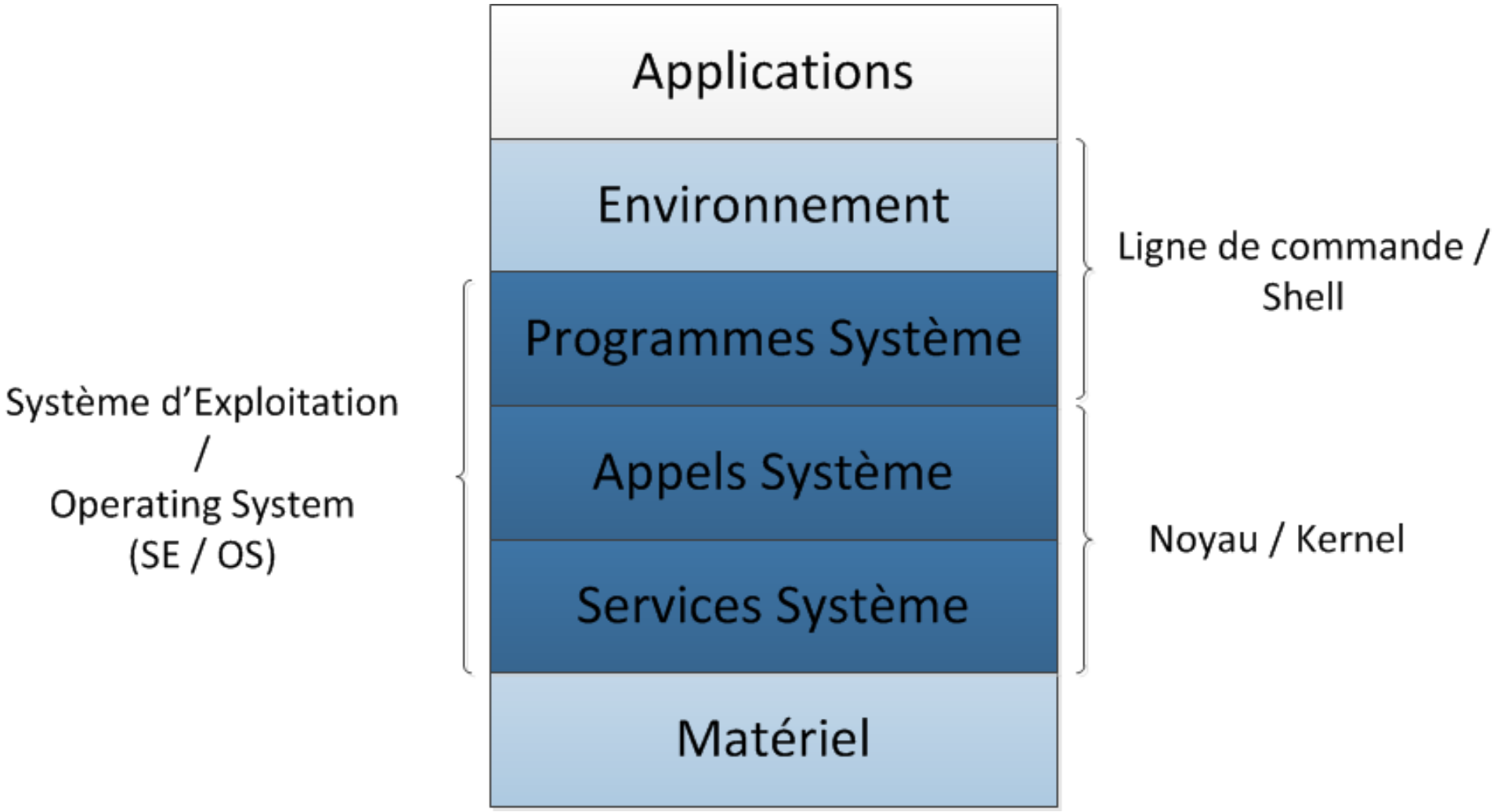
Applications Spécifiques
(au matériel)



Les Systèmes d'Exploitation



Les Systèmes d'Exploitation



Références Bibliographiques

- FreeBSD Handbook
- Linux Man Pages
- The Open Group
  Single UNIX Specification / SUSv4

Usage des Manuels

- Commande `man`
- Plusieurs sections :
  - 1 commande/programme
  - 2 appel système (syscall)
  - 3 fonction C (subroutine)
  - 4 fichiers spéciaux
  - 5 format de fichier
  - 7 macros et conventions
  - 8 commande de maintenance (super user)

Usage des Manuels

- Par défaut : man 1

`man passwd` = `man 1 passwd`

`man 5 passwd` = format du fichier passwd

`man printf` != `man 3 printf`

`man read` != `man 2 read`

Usage des Manuels

- Pour lire le manuel de la commande : `ls`

```
man ls
man 1 ls
```

- Pour lire le manuel de l'appel system : `read`

```
man 2 read
```

- Pour lire le manuel de la fonction C : `printf`

```
man 3 printf
```

Shell

- Interpréteur de commandes
  - Lit ce que l'on tape/Exécute ce qu'il comprend
  - Permet de lancer des programmes…
  - …et de les enchaîner les uns après les autres
- Récupère les messages des programmes
  - Et les renvoie vers le terminal
- Command Line Interface (CLI - opposé de GUI)
- Shell != Terminal

Shell

- Famille **sh** :
  - sh (Bourne Shell)
  - bash (Bourne Again Shell)
  - ksh (Korn Shell) <= utilisé dans environnement certifié
  - pdksh (Public Domain ksh)
  - ash / dash
  - zsh

- Famille **csh** :
  - csh (C Shell - disparu)
  - tcsh (Tenex C Shell)

Terminal

- Fenêtre « noire » (ou verte)
  - Envoie les caractères lus vers le programme contenu (le shell en général)
- Transfère les caractères entre :
  - Clavier => Ordinateur => Programme
  - Programme => Ordinateur => Écran
- Plusieurs modes possibles de fonctionnement
- Codes de caractères dépendent de la langue configurée

  0100.0001 = A en ascii US… mais et en UTF-8 ?

Liste rapide d'outils classiques

- Commandes/Outils de base :
  - man, sh, bash, tcsh, exit, echo
  - cd, ls, cp, mv, rm, mkdir, rmdir
  - vi, vim, emacs, nano, ed

- Outils :
  - find, date,
  - cat, cut, paste, tr, tee, sort, mktemp, basename, …
  - diff, head, tail, more, wc

- Outils avancés :
  - grep, expr, sed, awk, ed

Utilitaires

- man
- sh, bash
- ls, cp, mv, rm, mkdir, rmdir, touch, pwd, mktemp, …
- cat, cut, paste, tr, tee, wc, sort, head, tail, more, ed, diff, dirname, basename, …
- find, grep, expr, sed, awk
- emacs, vi, vim, nano

Programmes Système

- sh, bash, tcsh
- ls, cp, mv, rm, mkdir, rmdir, touch, pwd, du, df, which, dirname, basename, chmod, umask, …
- date, env, ed
- ps, top, kill, who, whoami

Builtins Shell

- Redirections : < > << >> |
- test / [
- echo
- exit
- set / unset
- export (setenv)
- alias / unalias
- read
- readonly
- return
- eval
- let
- type
- =
- cd
- source
- jobs
- fg / bg
- fc
- break / continue
- kill
- command
- dirs / pushd / popd
- times
- wait

Langage Shell

- Tout script commence par :
  `#! /bin/sh`

- if, then, elif, else, fi
- case, in, esac
- while, do, done
- until, do, done
- for, in, do, done

- mode maths

- shift
- break
- read

Langage Shell : echo

- echo
- Affiche une chaîne de caractères dans le terminal

```
echo "Hello World!"
```

- echo -e interprète les caractères spéciaux

```
echo -e "Hello World!\nBonjour tout le monde !"
```

\n sera interprété comme un retour à la ligne

Langage Shell

- Tout script commence par :
  `#! /bin/sh`

- Si une ligne est trop longue, on peut la couper par un '\'
  - En pratique, on tape \ puis on appuie sur ENTER

```
echo "Ceci est une énorme ligne" \
"avec beaucoup trop de caractères"
```

Langage Shell : Variables

- Commencent par un `$`
  - Ex : `$var`

- En cas de variable avec un nom pouvant ressembler à une commande, on encadre le nom de la variable par des `{ }`
  - Ex : `$thetime` => `${thetime}` ou `${the}time`

Langage Shell : Variables

- Déclaration :

```
MaVariable="Une Valeur"
```

- Lecture :

```
echo $MaVariable
echo ${MaVariable}

echo Bloub${MaVariable}Blob
```

Langage Shell : Variables

- Les variables « classiques » sont limitées au shell courant
  - Testez ce code :

```
var="allo"
echo $var
bash
echo $var
```

Langage Shell

- Paramètres du script :

```
$#   $0   $1 $2 $3 $4 $5 $6 $7 $8 $9
```

- Paramètres spéciaux :

```
      $#   $*   $@   $$   $?   $-   $!
```

- Commentaires :

```
# commentaire
```

Langage Shell : paramètres spéciaux

$0	Commande/Shell actuel
$-	Paramètres du shell actuel
$#	Nombre de paramètres positionnels
$*	Liste des paramètres en un seul mot
$@	Liste des paramètres en plusieurs mots
$?	Valeur de retour de la dernière commande
$!	PID du dernier processus en background
$$	PID du shell courant

Langage Shell : arguments

- Testez :

```
./script.sh "allo" 42 "bloub"
./script.sh "coucou"
```

(contenu de script.sh)

```
#! /bin/sh

echo $#      # Affiche le nombre d'arguments
echo $1      # Affiche l'argument 1
echo $2      # Affiche l'argument 2
echo $3      # Affiche l'argument 3
```

Langage Shell : set/unset

- Les paramètres positionnels peuvent être modifiés pendant l'exécution :
  - man set(1) unset(2)

```
set 'ls -la' 'bla' 'pouet'
echo $1
echo $2
echo $3
```

set contrôle beaucoup de paramètres liés au shell.
L'affichage du script en cours : set -x
Ou beaucoup d'autres choses.

Langage Shell : set/unset

- Vider le contenu d'une variable :
  `var=`

- Libérer une variable :

```
var=42
echo $var
unset var
echo $var
```

Langage Shell : readonly variable

- Déclarer une variable en lecture seule :
  `readonly var="Test"`

- Impossible de libérer les variables readonly
  …sans passer par du bas niveau qui ne fonctionnera pas toujours…

Langage Shell : if/then/else/fi

- if, then, else, fi

```
if [ "$#" = 0 ]
then
    echo "Error"
else
    echo "OK"
fi

# sans ;
```

```
if [ "$#" = 0 ]; then
    echo "Error"
else
    echo "OK"
fi


# avec ;
```

Langage Shell : if/then/elif/else/fi

- if, then, elif, else, fi

```
if [ "$#" = 0 ]; then
   echo "Error"
elif [ "$#" = 1 ]; then
   echo "1 Param"
else
    echo "OK"
fi
```

Langage Shell : if/then/elif/else/fi

- if, then, elif, else, fi

```
if [ "$#" = 0 ] && [ "A" = "A" ]
then
   echo "OK"
fi
```

Langage Shell : if/then/elif/else/fi

- if, then, elif, else, fi

```
if [ "$#" = 0 ] || [ "B" = "A" ]
then
   echo "OK"
fi
```

Langage Shell : case/in/esac

- case/in/esac

```
case "$1" in
   "r" )
            echo "couleur Rouge" ;;
   "g" )
            echo "couleur Verte" ;;
   "b" )
            echo "couleur Bleue" ;;
   * )
            echo "couleur inconnue" ;;
esac
```

Langage Shell : while/do/done

- while, do, done (tant que condition vraie)

```
i=0
max=3

while [ $i -ne $max ]
do
    echo "Test : $i"
    i=$(( $i + 1 ))
done
```

```
i=0
max=3

while [ $i -ne $max ]; do
    echo "Test : $i"
    i=$(( $i + 1 ))
done
```

Langage Shell : until/do/done

- until, do, done (tant que condition pas vraie)

```
OK="NO"

until [ "$OK" = "OK" ]
do
    echo "Voila"
    OK="OK"
done
```

```
OK="NO"

until [ "$OK" = "OK" ]; do
    echo "Voila"
    OK="OK"
done
```

Langage Shell : for/in/do/done

- for, in, do, done

```
for w in "$@"
do

  echo "Mot : $w"

done
```

```
for w in "$@" ; do

  echo "Mot : $w"

done
```

Langage Shell : $@ et $*

- Testez donc :
  ./script.sh a b c
  ./script.sh ‘a b’ c

```
#! /bin/sh

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

Puis avec :
./script.sh a b c
./script.sh ‘a b’ c

```
#! /bin/sh

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

Langage Shell : $@ et $*

- Testez donc :
  ./script.sh a b c
  ./script.sh ‘a b’ c

```
#! /bin/sh

for i in "$*"
do
    echo $i
done
```

Puis avec :
./script.sh a b c
./script.sh ‘a b’ c

```
#! /bin/sh

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

Langage Shell : for et plus…

- For peut itérer sur beaucoup de choses…
  (en fait l'* est transformée par les fichiers autour du dossier courant)

```
#! /bin/sh

for f in *; do
    echo "File -> $f"
done
```

Langage Shell : for et plus…

- For itère en fait sur une liste de paramètres qu'on lui fournit. La commande `seq` permet de générer la liste des nombres sur lesquels itérer :

  ./script.sh 12

```
#! /bin/sh

for i in `seq 0 $1`
do
    echo $i
done
```

Langage Shell : shift

- Instruction `shift` décale les paramètres positionnels

Testez :

./script.sh param1 param2 param3

```
#! /bin/sh

echo $#
echo $1
echo $*
shift

echo $#
echo $1
echo $*
shift

echo $#
echo $1
echo $*
```

Langage Shell : break

- Instruction `break` stoppe une boucle en cours (for, while, until)

```
#! /bin/sh

i=0
max=3

while [ $i -ne $max ]; do
    echo "Test : $i"
    if [ $i -eq 1 ]; do
           break
    fi
    i=$(( $i + 1 ))
done
```

Langage Shell : read

- Instruction `read` lit du texte du terminal, et le met dans une variable

```
#! /bin/sh

var1=A
echo $var1
echo "Input Var2 :"
read var2
echo $var2
echo "Input Var1 :"
read var1
echo $var1
```

Langage Shell : true false

- Comme leurs noms l'indiquent, `true` et `false` correspondent aux valeurs « vrai » et « faux » habituelles

```
#! /bin/sh

if [ true ]; then
   echo "On passera toujours ici"
else
   echo "On ne passera jamais ici"
fi
```

Langage Shell : test

- test ou [
  - test -n *str* chaîne n'est pas vide OU inexistante
  - test -z *str* chaîne est vide OU inexistante

    (toujours mettre les variables entre guillemets)

  - test -e *file* fichier existe
  - test -d *file* fichier est un répertoire
  - test -x *file* fichier est-il exécutable
    (ou dossier, pour pouvoir le traverser)

Langage Shell : test

```
#! /bin/sh
str1=
str2=""
str3="Bla"

test -n $str1          # vrai
test -z $str1          # vrai
test -n $str2          # vrai
test -z $str2          # vrai
test -n $str3          # vrai
test -z $str3          # faux
test -n $str4          # vrai
test -z $str4          # vrai
```

Langage Shell : test

```
#! /bin/sh
str1=
str2=""
str3="Bla"

test -n "$str1"        # faux
test -z "$str1"        # vrai
test -n "$str2"        # faux
test -z "$str2"        # vrai
test -n "$str3"        # vrai
test -z "$str3"        # faux
test -n "$str4"        # faux
test -z "$str4"        # vrai
```

Langage Shell : mode maths

- 2 modes dans le shell :
  - Mode texte/Comparaisons classiques

    ```
    var="lol"
    if [ "$var" != "$b" ]; then
    if test "$var" != "$b" ; then
    ```

  - Mode maths

    ```
    i=42
    if [ $i -ne 42 ]; then

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

    <= addition mathématique

Langage Shell : mode maths

- Mode maths entouré par (( )) ou $(( ))

- Comparaisons mathématiques :
  - -eq, -ne, -gt, -ge, -lt, -le, =, != <= pour test ou [ ];
  - ==, !=, >, >=, <, <= <= pour (( ))

```
((nb = 4 + 3))
nb=$(( 4 + 3 ))

if (( nb < 42 )); then
```

Langage Shell : mode maths

- Mode maths entouré par (( )) ou $(( ))

- Opérateurs arithmétiques :

+	–	++	––	*	/	%
addition	soustraction	incrémentation	décrémentation	multiplication	division	reste

- Opérateurs logiques :

& &	| |	!
ET	OU	NON

Langage Shell : mode maths

```
#! /bin/sh

A=60
B=13

((B++))
C=$(( A * B ))

echo $A $B $C

if (( (C >= 42) && (A < 100) )); then
      echo "OK"
fi
```

Langage Shell : mode maths

- Mode maths entouré par (( )) ou $(( ))

- Opérateurs bit à bit :

&	|	^	~	>>	<<
ET binaire	OU binaire	XOR	Inversion des bits	Décalage à droite	Décalage à gauche

Pour A = 60 = % 0011 1100
et B = 13 = % 0000 1101

A & B	A | B	A ^ B	~A	A >> 2	A << 2
% 0000 1100	% 0011 1101	% 0011 0001	% 1100 0011	% 0000 1111	% 1111 0000

Langage Shell : mode maths

```
#! /bin/sh

A=60
B=13

C=$(( A & B ))
echo "A & B : "$C

C=$(( A | B ))
echo "A | B : "$C

C=$(( A ^ B ))
echo "A ^ B : "$C
```

```
#! /bin/sh

A=60
B=13

C=$(( ~A ))
echo "~A : "$C

C=$(( A >> 2 ))
echo "A >> 2 : "$C

C=$(( A << 2 ))
echo "A << 2 : "$C
```

Langage Shell : strings

- Opérations sur les chaînes de caractères intégré à la gestion des variables :

${variable}	*Affiche la variable*	*mmachainee*
${#variable}	Longueur de la chaîne (strlen)	10
${variable%a*}	Retire le plus petit suffixe possible	mmach
${variable%%a*}	Retire le plus grand suffixe possible	mm
${variable#*m}	Retire le plus petit préfixe possible	machainee
${variable##*m}	Retire le plus long préfixe possible	achainee
TMP='a*'		
${variable%$TMP}	Retire plus petit suffixe avec variable	mmach
TMP='*m'		
${variable#$TMP}	Retire plus petit préfixe avec variable	machainee

Langage Shell : strings ${#VAR}

```
#! /bin/sh

VAR="mmmamachainedecaracteresblablablabla"
echo -e '$VAR '"\t\t"$VAR

echo ""

# len
echo -e '${#VAR} '"\t"${#VAR}
```

Langage Shell : strings ${VAR%xxx}

```
# Remove Smallest Suffix Pattern
echo -e '${VAR%bla} '"\t"${VAR%bla}

# Remove Largest Suffix Pattern
echo -e '${VAR%%bla} '"\t"${VAR%%bla}

TMP='a*'
echo '$TMP '$TMP
echo -e '${VAR%$TMP} '"\t"${VAR%$TMP}
echo -e '${VAR%%$TMP} '"\t"${VAR%%$TMP}
```

Langage Shell : strings ${VAR#xxx}

```
# Remove Smallest Prefix Pattern
echo -e '${VAR#m} '"\t"${VAR#ma}

# Remove Largest Prefix Pattern
echo -e '${VAR##m} '"\t"${VAR##ma}

TMP='*m'
echo '$TMP '$TMP
echo -e '${VAR#$TMP} '"\t"${VAR#$TMP}
echo -e '${VAR##$TMP} '"\t"${VAR##$TMP}
```

Langage Shell : strings

- Opérations sur les chaînes de caractères intégrées à la gestion des variables :

TMP='bla'	
${variable}	
${variable:-$TMP}	**Use Default Values**
${variable-$TMP}	
${variable:=$TMP}	**Assign Default Values**
${variable=$TMP}	
${variable:?$TMP}	**Indicate Error if Null or Unset**
${variable?$TMP}	
${variable:+$TMP}	**Use Alternative Value**
${variable+$TMP}	

Langage Shell : strings

- Opérations sur les chaînes de caractères intégrées à la gestion des variables :

TMP='bla'	
${variable}	*Affiche la variable*
${variable:-$TMP}	Si *VARIABLE* est nulle ou unset affiche *bla*, sinon affiche $variable
${variable-$TMP}	Si *VARIABLE* est nulle affiche **rien**, si unset affiche *bla*, sinon $variable
${variable:=$TMP}	Si *VARIABLE* est nulle ou unset affiche *bla*, sinon $variable
${variable=$TMP}	Si *VARIABLE* est nulle ou unset affiche *bla*, sinon $variable
${variable:?$TMP}	Si *VARIABLE* est nulle ou unset ERREUR, sinon $variable
${variable?$TMP}	Si *VARIABLE* est nulle affiche **rien**, si unset ERREUR, sinon $variable
${variable:+$TMP}	Si *VARIABLE* est nulle ou unset affiche **rien*, sinon affiche *bla*
${variable+$TMP}	Si *VARIABLE* est unset affiche **rien*, sinon affiche *bla*

Langage Shell : strings ${VAR:-xxx}

```
#! /bin/sh

VAR="chaine"
echo '$VAR '$VAR
echo -e '${VAR:-PLOP} '"\t"${VAR:-PLOP}
echo -e '${VAR-PLOP} '"\t"${VAR-PLOP}

VAR=""
echo '$VAR *empty*'
echo -e '${VAR:-PLOP} '"\t"${VAR:-PLOP}
echo -e '${VAR-PLOP} '"\t"${VAR-PLOP}

unset VAR
echo '$VAR unset'
echo -e '${VAR:-PLOP} '"\t"${VAR:-PLOP}
echo -e '${VAR-PLOP} '"\t"${VAR-PLOP}
```

Langage Shell : strings ${VAR:=xxx}

```
#! /bin/sh

VAR="chaine"
echo '$VAR '$VAR
echo -e '${VAR:=PLOP} '"\t"${VAR:=PLOP}
echo -e '${VAR=PLOP} '"\t"${VAR=PLOP}

VAR=""
echo '$VAR *empty*'
echo -e '${VAR:=PLOP} '"\t"${VAR:=PLOP}
echo -e '${VAR=PLOP} '"\t"${VAR=PLOP}

unset VAR
echo '$VAR unset'
echo -e '${VAR:=PLOP} '"\t"${VAR:=PLOP}
echo -e '${VAR=PLOP} '"\t"${VAR=PLOP}
```

Langage Shell : strings ${VAR:+xxx}

```
#! /bin/sh

VAR="chaine"
echo '$VAR '$VAR
echo -e '${VAR:+PLOP} '"\t"${VAR:+PLOP}
echo -e '${VAR+PLOP} '"\t"${VAR+PLOP}

VAR=""
echo '$VAR *empty*'
echo -e '${VAR:+PLOP} '"\t"${VAR:+PLOP}
echo -e '${VAR+PLOP} '"\t"${VAR+PLOP}

unset VAR
echo '$VAR unset'
echo -e '${VAR:+PLOP} '"\t"${VAR:+PLOP}
echo -e '${VAR+PLOP} '"\t"${VAR+PLOP}
```

Langage Shell : strings ${VAR:?xxx}

```
#! /bin/sh

VAR="chaine"
echo '$VAR '$VAR
echo -e '${VAR:?PLOP} '"\t"${VAR:?PLOP}
echo -e '${VAR?PLOP} '"\t"${VAR?PLOP}

VAR=""
echo '$VAR *empty*'
echo -e '${VAR:?PLOP} '"\t"ERROR     <= ${VAR:?PLOP}
echo -e '${VAR?PLOP} '"\t"${VAR?PLOP}

unset VAR
echo '$VAR unset'
echo -e '${VAR:?PLOP} '"\t"ERROR     <= ${VAR:?PLOP}
echo -e '${VAR?PLOP} '"\t"ERROR      <= ${VAR?PLOP}
```

Si VAR est vide ou unset, dans ces 3 cas cela crashera le script

Spécificités de Bash : substrings et case

- Opérations sur les sous-chaînes de caractères intégrées à la gestion des variables dans Bash :

TMP='MaChaine'	
${variable}	*Affiche la variable*
${variable:2}	Démarre la chaîne à partir de l'offset 2 (affichera : « Chaine »)
${variable: -4}	Démarre la chaîne par la fin (4 derniers caractères : « aine »)
${variable:2:3}	Affichera 3 caractères à partir de l'offset 2 (« Cha »)
${variable: -4:3}	Affichera 3 caractères à partir de l'offset -4 (« ain »)
${variable^^}	Affiche le contenu de la variable en « uppercase » (majuscules)
${variable,,}	Affiche le contenu de la variable en « lowercase » (minuscules)
${variable^^a}	*Transforme tous les 'a' en 'A'*
${variable,M}	*Transforme le premier 'M' en 'm'*

Spécificités de Bash : substrings et case

```
#! /bin/bash

VAR=MaChaine

echo '${VAR:2} : '${VAR:2}
echo '${VAR:2:3} : '${VAR:2:3}

echo '${VAR: -4} : '${VAR: -4}
echo '${VAR: -4:3} : '${VAR: -4:3}

echo '${VAR^^} : '${VAR^^}
echo '${VAR,,} : '${VAR,,}

echo '${VAR,M} : '${VAR,M}
echo '${VAR^^a} : '${VAR^^a}
```

Spécificités de Bash : pattern matching

- ${variable/pattern/replace}

- On cherche un « pattern » dans la ${variable}, et s'il existe, on le remplace par « replace »
  - Une seule fois

- Si pattern démarre par :
  - / : on remplace autant de fois que nécessaire le pattern
  - # : on cherche uniquement à partir du début de la variable
  - % : on chercher uniquement à partir de la fin de la variable

Spécificités de Bash : pattern matching

```
#! /bin/sh

VAR="MaChaine"

echo '${VAR/"C"/"S"} : ' ${VAR/"C"/"S"}

echo '${VAR/a/i} : ' ${VAR/a/i}
echo '${VAR//a/i} : ' ${VAR//a/i}

echo '${VAR/#C/S} : ' ${VAR/#C/S}
echo '${VAR/#M/L} : ' ${VAR/#M/L}
echo '${VAR/#Ma/Lo} : ' ${VAR/#Ma/Lo}

echo '${VAR/%C/S} : ' ${VAR/%C/S}
echo '${VAR/%e/o} : ' ${VAR/%e/o}
echo '${VAR/%ne/po} : ' ${VAR/%ne/po}
```

Langage Shell : guillemets/quotes

- Double quote : " (sur AZERTY :) (touche 3)
  - Contient du texte et des variables remplacées
    `"Coucou $nom !"` => `"Coucou BOISSIER !"`
- Simple quote : ' (touche 4)
  - Contient du texte et des variables « non » remplacées
    `'Coucou $nom !'` => `'Coucou $nom !'`
- Back quote : ` (AltGr + 7)
  - Contient une commande qui sera exécutée dans un sous shell
    `` `ls -a` `` => `" . .. file1"`

Langage Shell : guillemets/quotes

- Variables imbriquées et parenthèses

```
NOM="BOISSIER"

VAR="Coucou ${NOM}"
echo ${VAR}

VAR='Coucou ${NOM}'
echo ${VAR}
```

Langage Shell : guillemets/quotes

- Inclusion des guillemets dans un texte :
  - on échappe avec \ , ou on change de guillemets
  - Impossible d'inclure des guillemets « simple » dans des guillemets simples (ils annulent déjà toute interprétation du contenu)

```
echo "Test 1 : \" ."
echo "Test 1 : ' ."

echo 'Test 2 : '\'' .'
echo 'Test 2 : " .'
```

Langage Shell : échappement

- Pour afficher certains caractères :
  - on échappe avec \
  - on utilise des simples quotes

****	**\$**	**\n**	**\r**	**\t**
Backslash	Symbole $	Retour à la ligne	Retour chariot	Tabulation horizontale
\v	\a	\b	\f	
Tabulation verticale	(bell)	Backspace	« Form feed »	

```
echo "Test : \\"
```

Langage Shell : fonctions

- Définition de fonctions proche du C
  - `function ma_fonction { }`
  - `ma_fonction() { }`
- Attention : exit quitte le script
  return quitte la fonction ($?)
- Pas de déclaration de paramètre explicite
- Accès aux paramètres par $1, $2, …
  - $0 reste le nom du script

Langage Shell : fonctions

```
#! /bin/sh

ajouter() {
      VAR=$(( $1 + $2 ))
      return $VAR
}

ajouter 3 2
RES=$?

echo $RES
```

```
#! /bin/sh

ajouter()
{
      VAR=$(( $1 + $2 ))
      return $VAR
}

ajouter 3 2
RES=$?

echo $RES
```

Langage Shell : strings & fonctions

```
#! /bin/sh

myfunc() {
  reqsubstr="$1"
  shift
  string="$@"
  if [ -z "${string##*$reqsubstr*}" ] ;then
    echo "String '$string' contains : '$reqsubstr'.";
  else
    echo "String '$string' doesn't have: '$reqsubstr'."
  fi
}

myfunc "a" "Ma Chaine"
myfunc "b c d" "Ma Chaine"
```

Langage Shell : source

- source : lit un script dans le shell courant et l'exécute (similaire à `sh script.sh`, mais sans ouvrir de sous-shell)

```
source script.sh

source autre_script.sh arg1 arg2
```

Langage Shell : eval

- eval : exécute une commande précédemment construite

```
num=42
cmd=num
echo $cmd

var='$'$cmd
echo $var

eval var='$'$cmd
echo $var
```

Langage Shell : Globbing

- Globbing, similaire aux RegExp…

?	1 caractère
*	0 ou n caractères
[aeiouy]	1 seul caractère parmi la liste
[m-z]	1 caractère compris entre m et z
[^ACTG] ou [!ACTG]	1 caractère différent de ceux de la liste

Langage Shell : Globbing

- Globbing, similaire aux RegExp…
- Énumère les fichiers ou dossiers correspondants à l'expression

```
echo /[a-e]??    # 3 caracteres
                 # /bin /dev /etc

echo /h*e        # N caracteres
                 # /home
```

Shell & Environnement

Qu'est-ce qui différencie un programme que j'exécute d'un programme exécuté par quelqu'un d'autre sur la même machine au même moment ?

- Le PID…
- L'utilisateur/UID…

Shell & Environnement

- L'OS différencie les sessions avec divers ID
  - Ex: UID et PID ont des valeurs uniques
- Toutes ces variables à valeurs « uniques » forment : l'environnement
- Chaque utilisateur peut connaitre son environnement avec les commandes : `env` **ou** `set`

Shell & Environnement : Variables

- Pour définir une variable d'environnement :
  - Famille bourne shell :
    ```
    export MaVariable="valeur"
    unset MaVariable
    ```
  - Famille Csh :
    ```
    setenv MaVariable "valeur"
    unsetenv MaVariable
    ```
  - En C :
    man setenv(3) getenv(3) unsetenv(3)

Shell & Environnement : Variables

- Multiples variables d'environnement existantes :

Prompt : PS1, PS2, PS3, PS4

PATH, HOME, USER, PWD, HOSTNAME, LANG, EDITOR, SHELL, SHLVL, …

Shell & Environnement : IFS

- IFS : Internal Field Separator
- Séparateur de « paramètres »
  - Comment détecter le 1er, 2e, 3e, … paramètres ?
  - Par les espaces entre les mots…
  - …ou les tabulations.
- Liste des caractères d'espacement dans la variable IFS
  - Par défaut : <space> <tab> <newline>
  - Modifiable pour y mettre son propre séparateur

Shell & Environnement : alias

- Possibilité de créer des alias aux commandes
- Au lieu de taper `ls -la`, juste taper `ls`

```
alias ls="ls -la"
```

- Comment utiliser la commande originale en outrepassant les alias ?
- Avec un backslash devant :

```
\ls
```

Shell & Environnement : cd

- cd

- La commande « cd » n'est pas un programme
- On modifie seulement la variable d'environnement PWD avec cd !
- `cd dir` ⇔ `export PWD=${PWD}/dir`

- `cd -` on revient dans le dossier précédent
- `cd` on revient dans son *home directory*

Shell : Redirections de Flux

- 4 types de redirections :
  - Flux de Sortie

    `>` : écrasement fichier sortie & écriture

    `>>` : ajout en fin de fichier

  - Flux d'Entrée

    `<` : lecture fichier

    `<<` : « here doc » / lecture clavier jusqu'à un pattern

Shell : Redirections de Flux

> écrasement fichier sortie & écriture

```
$ cat file1
texte
$ ls -a > file1
$ cat file1
.     ..    file1
```

Shell : Redirections de Flux

>> ajout en fin de fichier

```
$ cat file1
texte
$ ls -a >> file1
$ cat file1
texte.     ..    file1
```

Shell : Redirections de Flux

< lecture fichier

```
$ cat file1
b
c
a

$ sort < file1
a
b
c
```

Shell : Redirections de Flux

<< lecture clavier jusqu'à un pattern (« here doc »)

```
$ sort << FIN
> b
> c
> a
> FIN

a
b
c
```

Shell : Flux Système

- « File Descriptors » liés aux processus

- 0, 1, 2
- STDIN, STDOUT, STDERR
- STandarD IN, STandarD OUT, STandarD ERRor

- Entrée Standard, Sortie Standard, Sortie d’Erreur

Shell : Redirections

2 / STDERR Sortie d'Erreur

```
$ ls . abcd
ls: impossible d'accéder à 'abcd': No such file or
directory
.:
file1       file2

$ ls . abcd 2> file2
.:
file1       file2
```

Shell : Redirections

1 / STDOUT Sortie Standard

```
$ ls . abcd
ls: impossible d'accéder à 'abcd': No such file or
directory
.:
file1       file2

$ ls . abcd 1> file2
ls: impossible d'accéder à 'abcd': No such file or
directory
```

Shell : Redirections

0 / STDIN Entrée Standard

```
$ cat file1
c
b

$ sort 0< file1
b
c
```

Shell : Redirections

>& Renvoyer un file descriptor vers un autre

```
$ ls . abcd
ls: impossible d'accéder à 'abcd': No such file or
directory
.:
file1       file2

$ ls . abcd > file1 2>&1
```

Shell : Redirections

>& Renvoyer un file descriptor vers un autre

(`X >& Y` on renvoie X vers Y)

```
$ ls . abcd
ls: impossible d'accéder à 'abcd': No such file or
directory
.:
file1       file2

$ ls . abcd 2>&1 > file1
ls: impossible d'accéder à 'abcd': No such file or
directory
```

Shell : Redirections

>&- Fermer un file descriptor

(`X >&-` on renvoie ferme X)

```
$ ls . abcd
ls: impossible d'accéder à 'abcd': No such file or
directory
.:
file1       file2

$ ls . abcd 2>&-
.:
file1       file2
```

Shell : Redirections

```
$ ls . abcd > file1 2> file2
```

file1 :

```
.     ..    file1      file2
```

file2 :

```
ls: impossible d'accéder à 'abcd': No such file
or directory
```

Shell : Redirections & exec

- exec : deux usages
  - Permet de lancer une commande SANS forker… c'est-à-dire : écrase le programme courant par un autre (comme en C)

```
var="tcsh"
exec $var
```

Shell : Redirections & exec

- exec : deux usages
  - Créer et rediriger des flux (les File Descriptors)

```
exec 3< file1          # Ouvre le fd 3 en
                       #   lecture pour file1
read -u 3 LINE         # Lit une ligne depuis
echo $LINE             #   fd 3 et l'affiche

read LINE <&3          # Autre façon de lire
echo $LINE             #   le fd 3

exec 3<&-              # Fermeture du fd 3
```

Shell : Redirections & exec

- exec : deux usages
  - Créer et rediriger des flux (avec File Descriptors)

Au début d'un script : (redirige stdout et stderr vers un fichier)

```
#! /bin/sh

exec >custom.log 2>&1
ps ; ls -42

exit
```

Attention : ne pas oublier de `exit` *à la fin (pour reprendre la main)*

Shell : Redirections & exec

`exec 3< file1`	Ouvre le fichier « file1 » en lecture vers le File Descriptor 3
`exec 4> file2`	Ouvre le fichier « file2 » en écriture depuis le fd 4
`exec 8<> file3`	(BASH ONLY) Ouvre file3 en lecture et écriture avec fd 8
`exec 6>> file4`	Ouvre file4 en mode « append » depuis le fd 6
`exec 5<&0`	Redirige la lecture du fd 0 vers le fd 5
`exec 7>&4`	# copy write fd 4 onto fd 7
`exec 3<&-`	# close the read fd 3
`exec 6>&-`	# close the write fd 6

Shell : Redirections & exec

```
#! /bin/sh

echo -e "Ligne 1\nLigne 2\nLigne 3" > file1

exec 3< file1           # open file1 in reading to 3

read -u 3 LINE          # read one line from fd 3
echo "echo1 : $LINE"

read LINE <&3           # read one line from fd 3
echo "echo2 : $LINE"

exec 3<&-               # close fd 3
```

Shell : Redirections & exec

```
#! /bin/sh

echo -e "Ligne 1\nLigne 2\nLigne 3" > file1

exec 3< file1              # open file1 in reading to 3
exec 4<&3                  # redirect read from 4 to 3

exec 3<&-                  # we can close fd 3

read -u 4 LINE             # read one line from fd 4
echo "echo1 : $LINE"

read LINE <&4              # read one line from fd 4
echo "echo2 : $LINE"

exec 4<&-                  # close fd 4
```

Shell : Redirections & exec

```
#! /bin/sh

echo "" > file2

exec 6> file2  # open file2 in writing to 6

echo "Test 1"
echo "Test 1" >&6   # write to fd 6

exec 6>&-      # close fd 6
```

Shell : Redirections & exec

```
#! /bin/sh

echo "" > file2

exec 6> file2           # open file2 in writing to 6
exec 7>&6               # redirect write from 7 to 6

exec 6>&-               # we can close fd 6

echo "Test 1"
echo "Test 1" >&7       # write to fd 7

exec 7>&-               # close fd 7
```

Shell : Redirections & exec

- exec : deux usages
  - Lance un programme (on indique le programme)
    `exec programme`
  - Redirige un flux (on n'indique pas de programme)
    `exec 2> file`

Shell : Redirections

- Attention dans tous les cas :

Ne jamais ajouter d'espace entre le numéro du file descriptor et l'opérateur

```
ls -42 2> error          # OK

ls -42 2 > error         # KO
```

Shell : Enchaînement

- Enchaîner plusieurs commandes

;	Enchaînement simple	Toutes les commandes sont effectuées
&&	Enchaînement conditionnel	Exécute chaque commande, de gauche à droite, tant que la valeur de retour est 0
||	Enchaînement conditionnel	Exécute chaque commande, de gauche à droite, tant que la valeur de retour n'est pas 0
&	Lancement en arrière plan/parallèle	Chaque commande est lancée en arrière plan (donc en parallèle)
|	Enchaînement avec passage de données	La sortie de la commande à gauche est reliée à l'entrée de celle à droite

Shell : Enchaînement

; Enchaînement simple

```
$ ls ; echo -e "\ntoto\n" ; ls
.:
.           ..          file1

toto
.:
.           ..          file1
```

Shell : Enchaînement

&& Enchaînement conditionnel (si return == 0)

```
$ ls && echo -e "\ntoto\n" && ls
.:
.           ..          file1

toto
.:
.           ..          file1
```

Shell : Enchaînement

&& Enchaînement conditionnel (si return == 0)

```
$ ls abcd && echo -e "\ntoto\n" && ls
ls: impossible d'accéder à 'abcd': No such file or
directory
```

Shell : Enchaînement

|| Enchaînement conditionnel (si return != 0)

```
$ ls abcd || echo -e "\ntoto\n" || ls
ls: impossible d'accéder à 'abcd': No such file or
directory

toto
```

Shell : Enchaînement

|| Enchaînement conditionnel (si return != 0)

```
$ ls || echo -e "\ntoto\n" || ls
ls: impossible d'accéder à 'abcd': No such file or
directory
```

Shell : Enchaînement

& Lancement en arrière plan (parallèle)

```
$ ls & echo -e "\ntoto\n" & ls
[1] 1337
[2] 1338
.:
.             ..            file1
.:
.             ..            file1

toto
[1]- Termine 2       ls
[2]+ Fini            echo -e "\ntoto\n"
```

Shell : Enchaînement

| Enchaînement avec passage de données

(pipe/tube : STDOUT est relié à STDIN entre chaque commande)

```
$ echo -e "b\nc\n1\n3\na\n2" | sort | grep "[a-z]"
a
b
c
```

Shell : bg / fg

- Background : tâches en arrière plan
- Foreground : tâche(s) visible(s)

- Pour lancer un programme en background : finir la commande par un &

```
$ emacs &
```

- Le programme fonctionne, mais n'est pas visible

Shell : bg / fg

- La commande `ps` permet de voir les processus en cours (dont ceux en arrière plan)
- La commande `jobs` permet de voir les processus en arrière plan du shell courant

- La commande `fg` permet de remettre en foreground une tâche en background

Shell : bg / fg

```
$ emacs &
[1] 3390

$ jobs
[1]+ En cours d'exécution  emacs &

$ fg
```

Shell : bg / fg

```
$ emacs & vi & top &
[1] 3390
[2] 3391
[3] 3392

$ jobs
[1]  En cours d'exécution  emacs
[2]+ En cours d'exécution  vi
[3]- En cours d'exécution  top

$ fg 2
```

=> vi revient au premier plan

Shell : bg / fg

- Appuyer sur Ctrl + Z pendant l'exécution d'un programme le stoppe, puis le met en arrière plan (signal SIGTSTP)

- Appuyer sur Ctrl + C pendant l'exécution d'un programme permet de l'arrêter, en général (signal SIGINT)

Shell : bg / fg

- Un processus actif stoppé par Ctrl + Z peut être relancé en premier plan avec `fg`

- Un processus actif stoppé par Ctrl + Z peut être relancé en arrière plan avec `bg`

Shell : bg / fg

```
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

int     main(void)
{
  int   time_wait = 4;
  char  *str1 = "Coucou1\n";
  char  *str2 = "Coucou2\n";

  write(STDOUT_FILENO, str1, strlen(str1));
  sleep(time_wait);
  write(STDOUT_FILENO, str2, strlen(str2));

  return (0);
}
```

Shell : wait

- wait

- Attends que des processus se terminent
  - Utile si des tâches sont lancées en fond
- Peut attendre des processus précis ou tous les précédents

```
sleep 5 &
wait && ls
```

Shell : wait

- wait
- wait *PID* attends que le processus n° *PID* se termine
- wait *JOB_ID* attends que le processus dans le shell courant avec le n° *JOB_ID* se termine

```
wait 4242
```

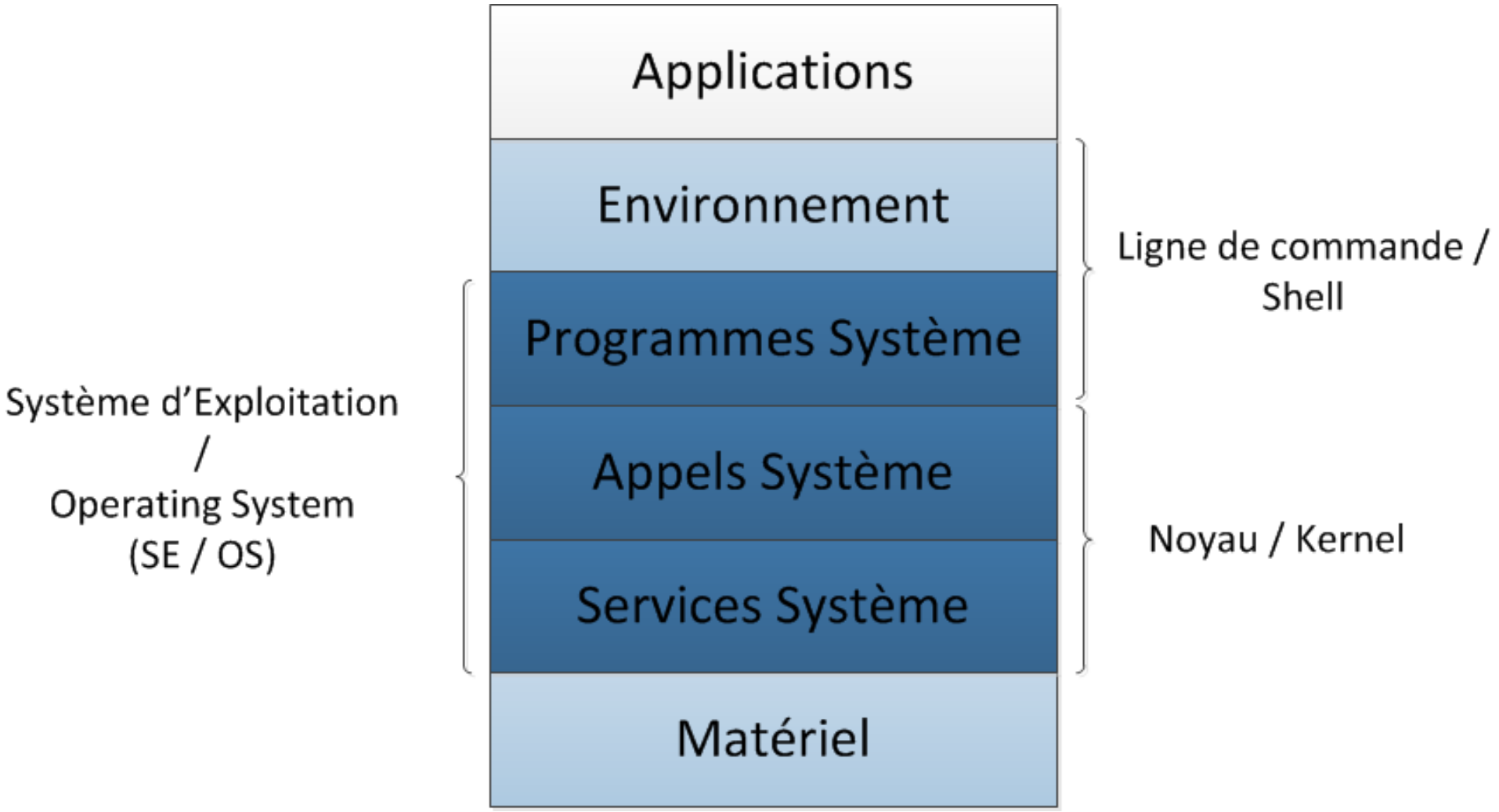
```
sleep 5 &
wait %1 && ls
```

Shell & Syscalls

- La plupart des commandes précédentes sont des surcouches « utilisateurs » pour utiliser des appels systèmes

- Rappel : un appel système (syscall) est une fonction que le système d'exploitation expose aux utilisateurs/développeurs, afin de permettre l'utilisation de la machine sans avoir à connaitre exactement le matériel

Shell & Syscalls



Shell & Syscalls

- Redirections :
  > et >> correspondent aux appels système
  - open(2)
  - close(2)
  - dup(2) dup2(2)

int fd = open(file1, O_TRUNC | O_CREAT |O_WRITEONLY);

int fd = open(file1, O_APPEND | O_WRITEONLY);

dup2(fd, 1);

Shell & Syscalls

- Enchaînements :
  ; && || & | correspondent à
  - fork(2)
  - exec(2) execvp(2) execve(2) exec…

```
if (fork())
    execvp(cmd);
else
    exit(0);
```

Shell & Syscalls

- Enchaînements :
  | correspond à
  - close(2)
  - dup(2) dup2(2)
  - fork(2)
  - exec(2) execvp(2) execve(2) exec…

Shell & Syscalls

- Background/Foreground :
  bg fg correspondent à
  - kill(2)

```
int pid = 1337;

kill(pid, SIGTSTP);

kill(pid, SIGCONT);
```

Shell & Syscalls

- Exec :
  - execve(2)
  - famille exec* (execv, execvp, execl, …)

```
int args = "-la";
execv("/bin/ls", args);
```

Shell & Syscalls

- Le Shell permet déjà à des non-développeurs d'utiliser les fonctionnalités typiquement unixiennes pour travailler

  ex : *fork, exec, dup, …*

- Le « paradigme » de l'OS peut donc malgré tout se percevoir bien au-delà des aspects développement système/bas niveau