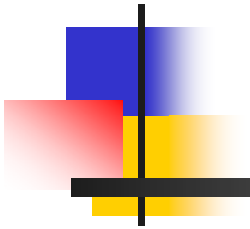


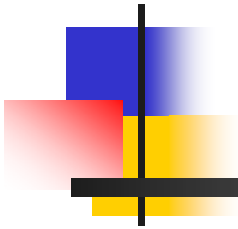
Université Grenoble Alpes

Le système Unix

Aspects utilisateur

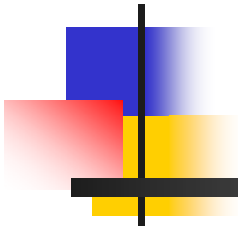
Jean-Michel Adam
UFR SHS

Damien Genthial
IUT de Valence



Introduction (1)

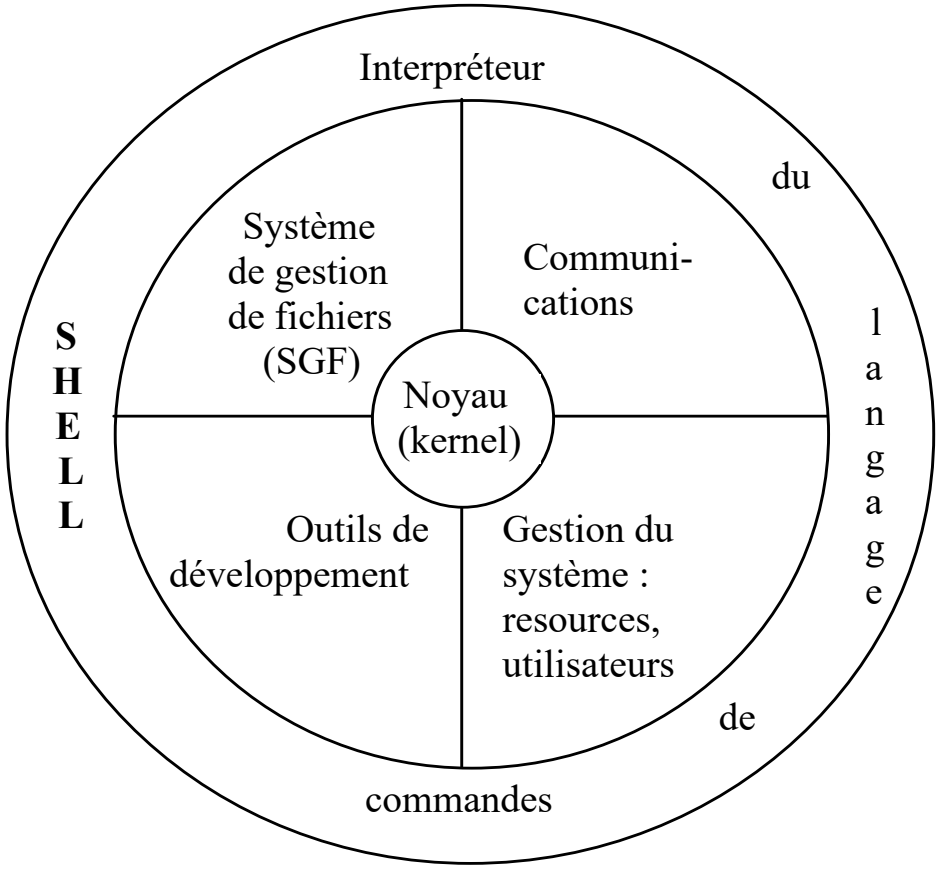
- Un peu d'histoire
  - Conçu pour des mini-ordinateurs au début des années 70 chez Bell
  - Conçu par des informaticiens, pour des informaticiens
  - Conçu pour être évolutif et ouvert
  - Grande diffusion : moyens et gros systèmes mais aussi petits systèmes : Linux, MacOS, Android)
- Les différentes versions
  - Versions constructeurs (IBM-AIX, HPUX, Ultrix, …)
  - BSD et dérivées (SunOS)
  - Linux
- Norme POSIX pour l'interface de programmation



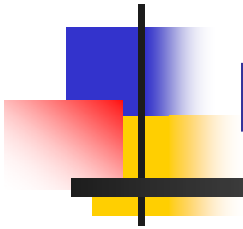
Introduction (2)

- Objectif du cours
  - Pratique d'Unix du point de vue utilisateur
  - Pratique de la programmation du langage de commande (shell)
- Bibliographie
  - **UNIX et Linux - Utilisation et administration**
    Jean-Michel Léry - Pearson Education – 3ème édition 2011
  - **Unix - Les bases indispensables**
    Michel Dutreix – ENI – 3ème édition 2015
  - **Unix: Programmation et communication**
    Rifflet J.M. et J.B. Yunès, Dunod 2003

Architecture générale d'Unix



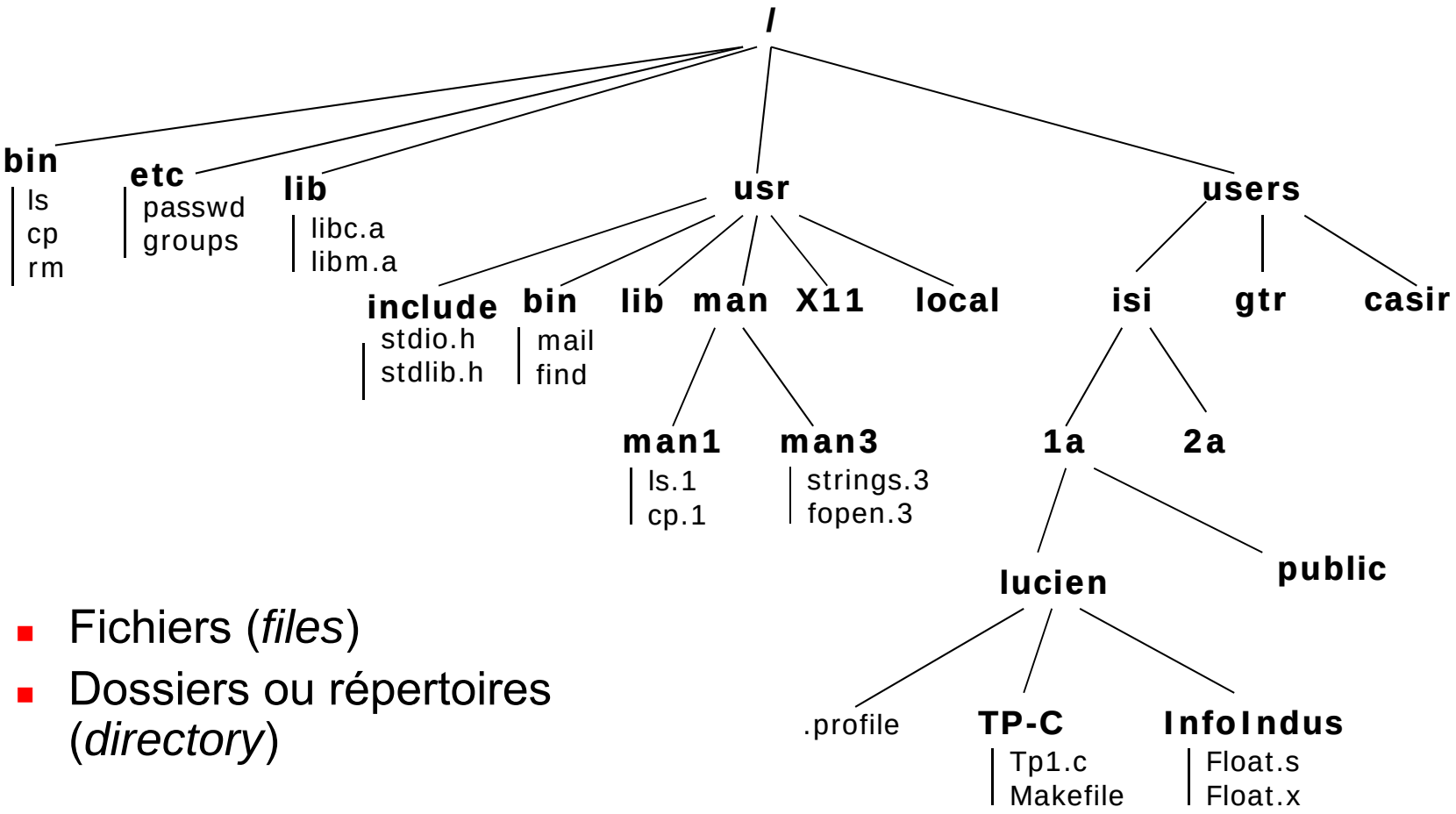
- Multi-tâches (multi-processus) et multi-utilisateurs
- Très grande facilité d'intégration en réseau
- Interface texte ou graphique



Plan du cours

- Le SGF : système de gestion de fichiers
  - Structure arborescente
  - Utilisateur et protections
  - Commandes de base
- Les processus
  - Principe, initialisation du système
- Le langage de commande
  - Généralités
  - Environnement et variables
  - Composition des commandes
  - Écriture de scripts : paramètres, structures de contrôle
  - Fonctions et procédures

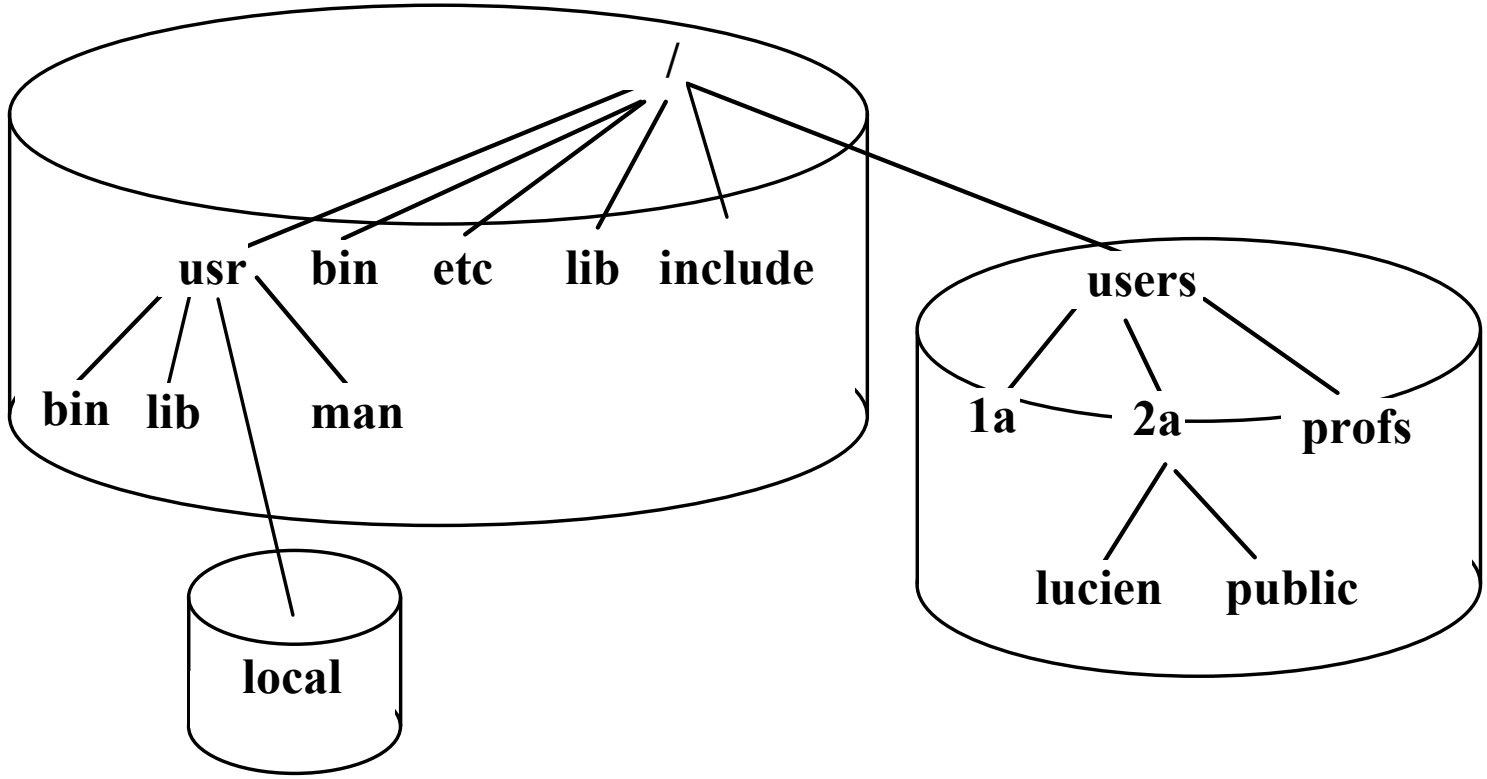
SGF : Structure arborescente unique



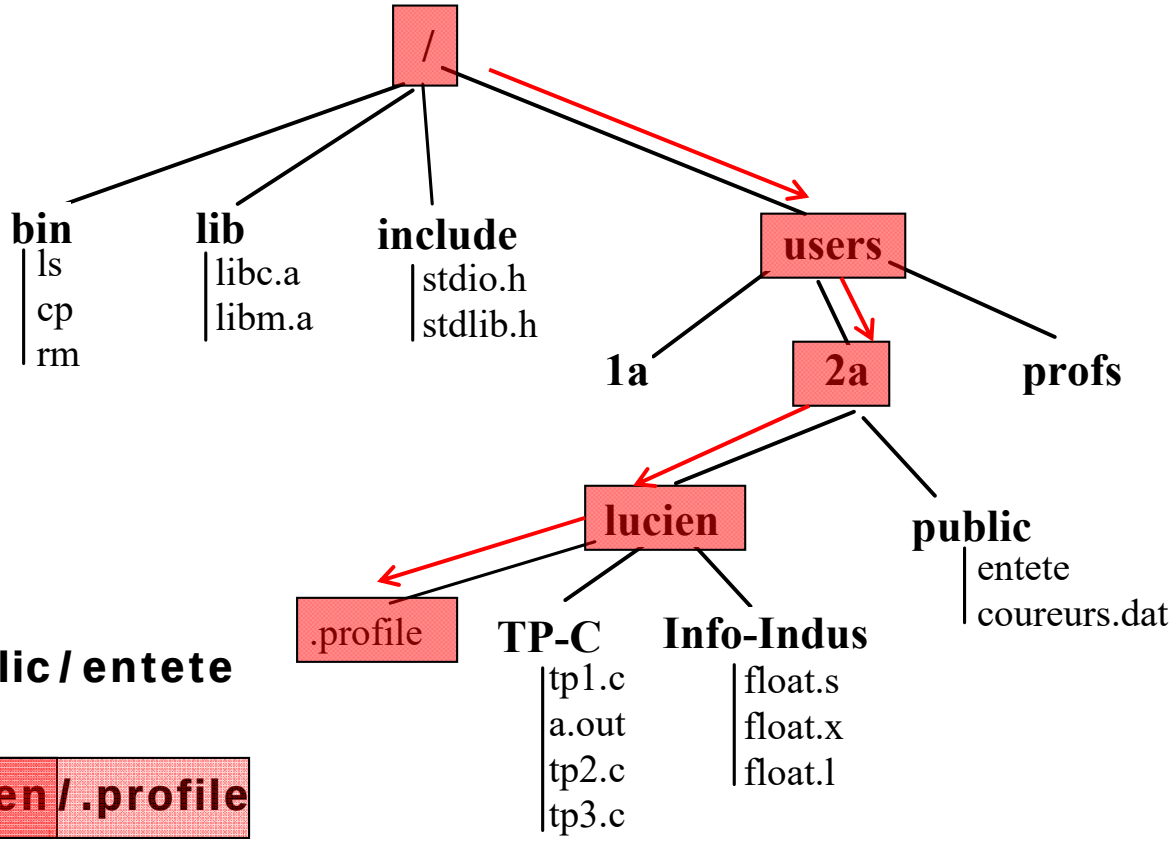
- Fichiers (*files*)
- Dossiers ou répertoires (*directory*)

SGF : Structure unique

- Vue logique indépendante de la réalité physique



Nom = chemin d'accès (PATH)



/bin/ls
/users/1a

/users/2a/public/entete

/users/2a/lucien/.profile

L'utilisateur dans l'arborescence

- Connexion
  - Nom d'utilisateur (identifiant ou *login*) + mot de passe
  - Bases de données des utilisateurs : **`/etc/passwd`** et **`/etc/groups`**

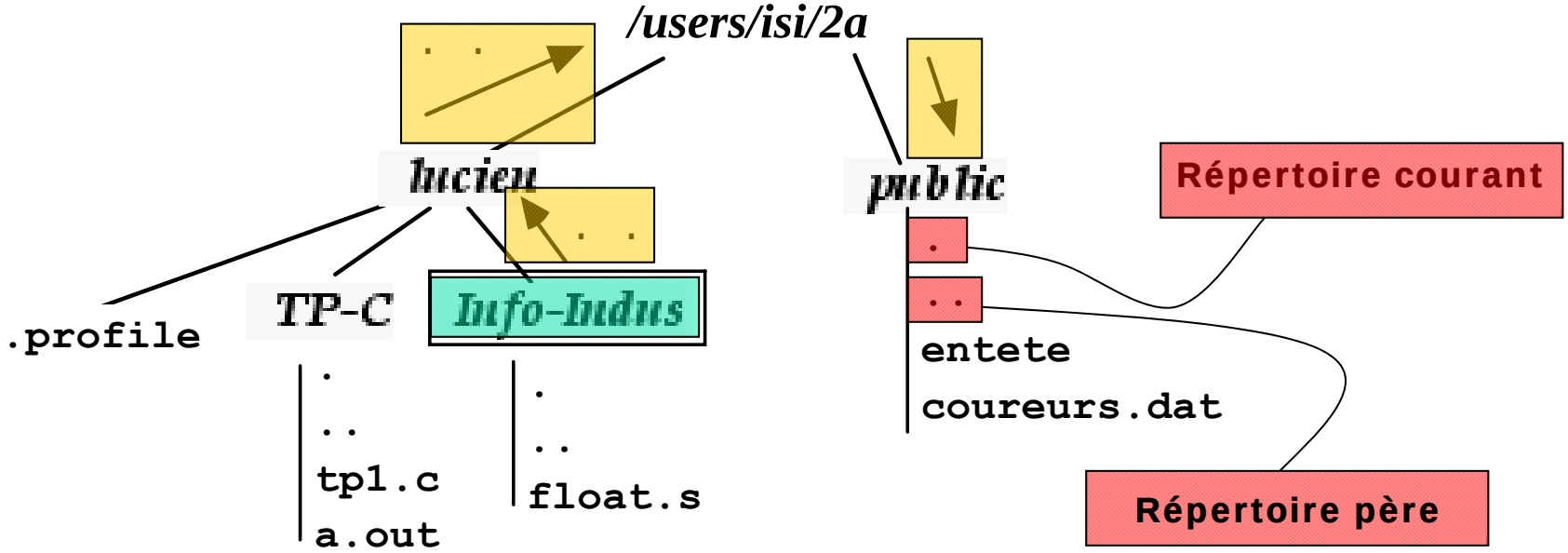
```
mdupont:x:1001:22:Marie Dupont Mass3:/users/mass3/mdupont:/bin/ksh
pdurand:x:2010:21:Pierre Durand:/users/staff/pdurand:/bin/ksh
```

- HOME SWEET HOME
  - Répertoire de travail (*working directory*)
  - Répertoire de travail par défaut (*home directory*)
- Noms absolus

```
/bin/ls
/users/mass3/mdupont/.profile
```

Noms relatifs

- Noms relatifs : ne commencent pas par /
  - Relatifs au répertoire courant (répertoire de travail)




```
float.s                  <=>   ./float.s
../TP-C/tp1.c            <=>   /users/isi/2a/lucien/TP-C/tp1.c
../../public/entete      <=>   /users/isi/2a/public/entete
```

SGF : caractères spéciaux

- Commencent par un point
  - Répertoire courant (`.`), répertoire père (`..`)
  - Fichiers de paramètres ou de configuration (répertoires)

    `.profile` `.bashrc`

    `.netscape` `.kde`
- Ne sont pas affichés par défaut
  - Utiliser `ls -a`
- Jokers (*wildcard characters*)

`*`	**Remplace n'importe quelle suite de caractères**
`?`	**Remplace exactement 1 caractère**
`[c1...cn]`	**Remplace exactement 1 des caractères c_1...c_n**
`[!c1...cn]`	**Remplace exactement 1 caractère sauf c1...cn**

SGF : commandes usuelles (1)

- Aide en ligne

  man <commande>

  - Ex :

    man ls affiche le manuel de **ls.**

    man man affiche le manuel de la commande **man.**

- Fichiers

  cat <fic> affiche le contenu du fichier <fic>

  more <fic> ...

  RC affiche la ligne suivante,

  SPACE affiche la page suivante,

  b affiche la page précédente,

  /<chaîne> recherche de **<chaîne>**

  h permet d'obtenir de l'aide,

  q ou **Ctrl-C** permet d'abandonner l'affichage.

SGF : commandes usuelles (2)

- Fichiers (suite)

  `lp <fic> ...` (*line printer*)

  `cp <fic1> <fic2>` (*copy*)

  `cp <fic> ... <rép>`

  `mv <fic1> <fic2>` (*move*)

  `mv <fic> ... <rép>`

  `rm <fic> ...` (*remove*)

  ⚡ ATTENTION, pas de récupération possible.

- Répertoires

  `ls [ options ] <rép> ...` (*list*)

  - Options intéressantes :
    - `-l` format *long*
    - `-a` tous les fichiers, y compris les cachés (*all*)
    - `-R` liste récursivement les sous-répertoires.

Démo

SGF : commandes usuelles (3)

- Répertoires (suite)

`pwd`	(*print working directory*)
`cd rép`	(*change directory*)
`mkdir rép ...`	(*make directory*)
`rmdir rép ...`	(*remove directory*)
`rm -r rép ...`	

⚡ ATTENTION, destruction récursive.

- Divers

```
clear, who, date, passwd (ou yppasswd)
mail -s <sujet du message> Destinataire@iut-valence.fr
<Texte du message (éventuellement sur plusieurs lignes)
.       (seul sur la ligne, ou Ctrl-D)
```

SGF : droits d'accès (1)

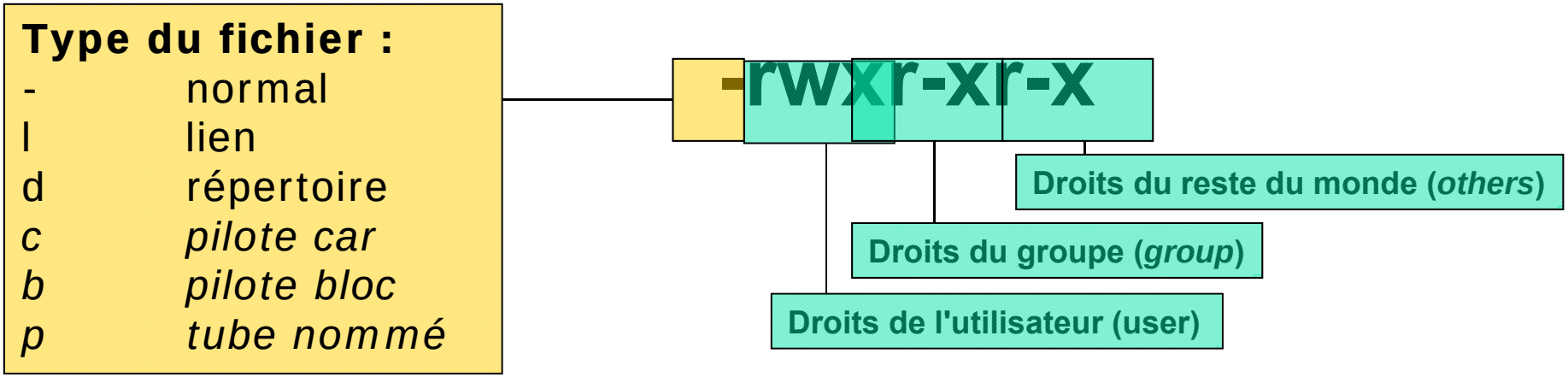
- Trois niveaux de protection : UGO

ls -l =>

-rw-r--r-- 1 lucien dciss 2566 oct 13 16:52 tp1.c

-rw-r--r-- 1 lucien dciss 223 oct 13 16:52 tp1.o

-rwxr-xr-x 1 lucien dciss 130 oct 13 16:49 tp1



SGF : droits d'accès (2)

- Trois modes d'accès : RWX
  - R : lecture (read), voir le contenu
  - W : écrire (write), modifier le contenu
  - X : exécuter (pour un fichier) ou traverser (pour un répertoire)
- Modification des droits

chown <propriétaire> <fic> ... *(change owner)*

chgrp <groupe> <fic> … *(change group)*

chmod <mode> <fic> ... *(change mode)*

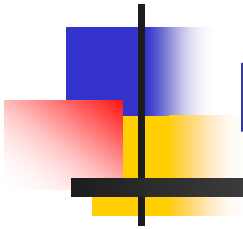
<mode> = 3 chiffres octaux (changement absolu)

chmod 755 monRep

<mode> = **[ugo][+-][rwx]** (changement relatif)

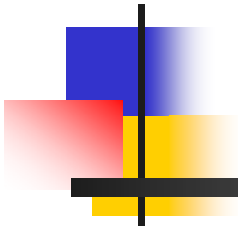
chmod go-w monRep

Démo



Plan du cours

- Le SGF : système de gestion de fichiers
  - Structure arborescente
  - Utilisateur et protections
  - Commandes de base
- Les processus
  - Principe, initialisation du système
- Le langage de commande
  - Généralités
  - Environnement et variables
  - Composition des commandes
  - Écriture de scripts : paramètres, structures de contrôle
  - Fonctions et procédures

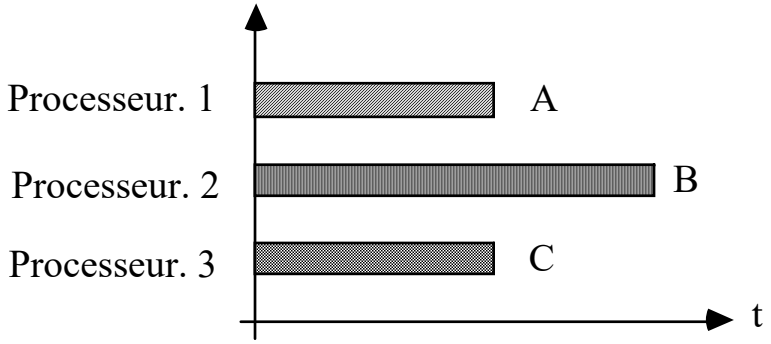


Processus : principe

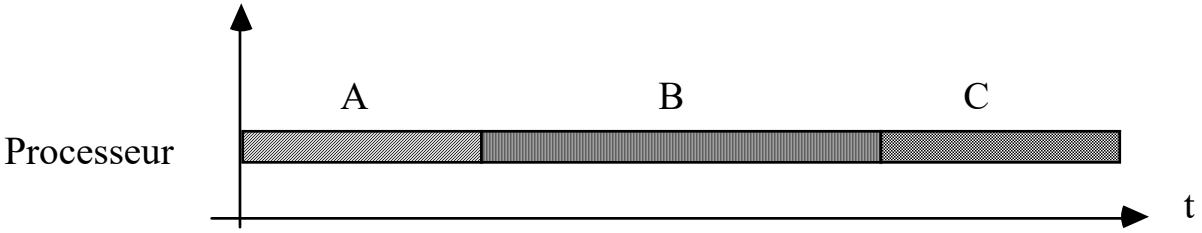
- Unix est multitâches et multi-utilisateurs
  - Tâche Unix = processus
- Processus = programme en cours d'exécution
  - Aspects dynamiques : évolution temporelle, état
  - Plusieurs processus peuvent exécuter le même programme (typiquement un éditeur de textes)
  - Une application peut être composée de plusieurs processus
- Exécution simultanée de plusieurs processus
  - Pb : avec un seul processeur ?
  - Utilisation du temps partagé

Processus : temps partagé

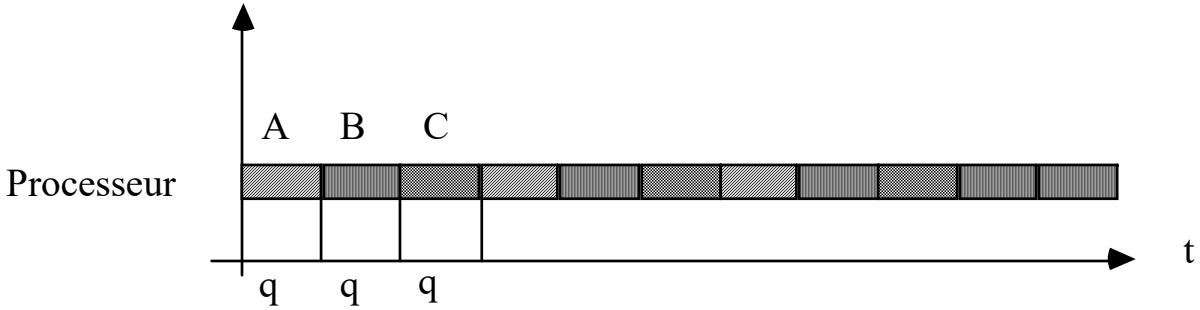
- Machine multiprocesseurs



- Monoprocesseur, monotâche



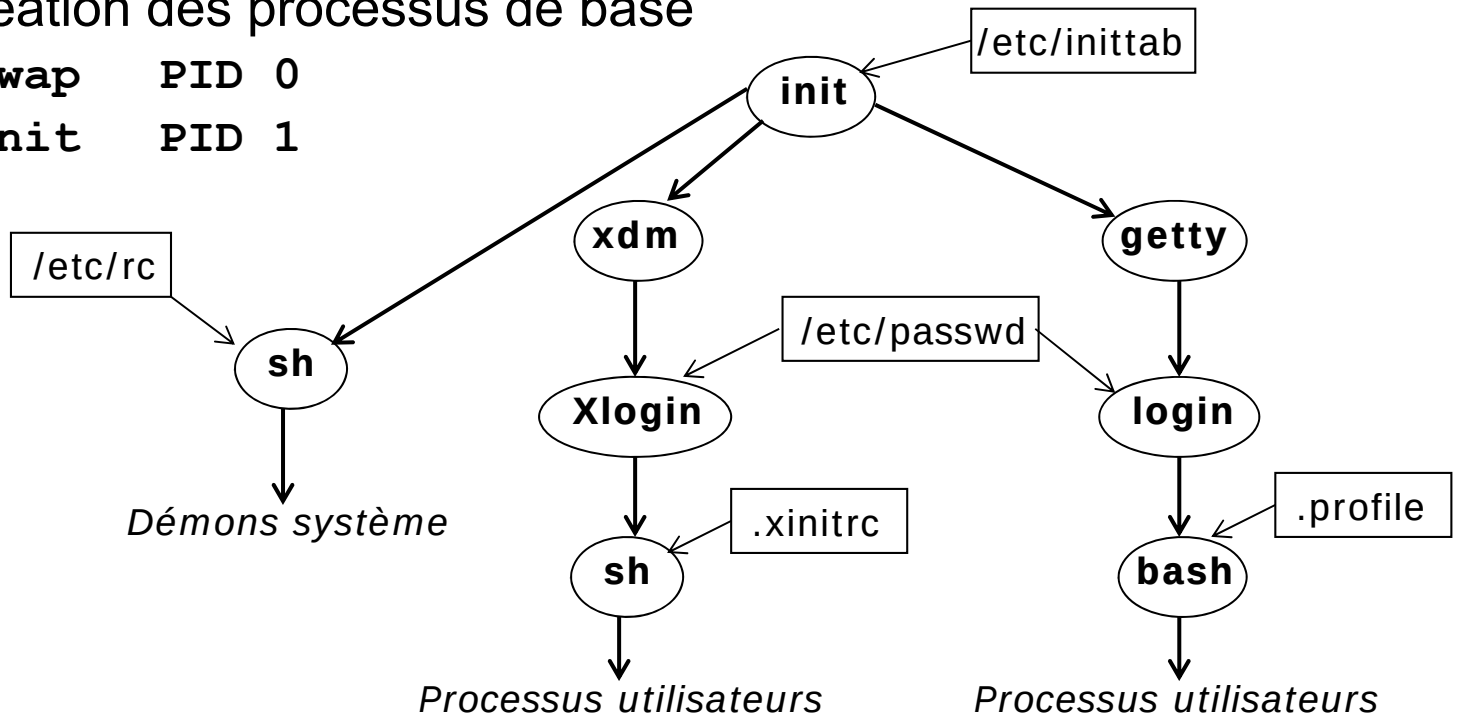
- Monoprocesseur, multitâches



Processus : cas d'Unix

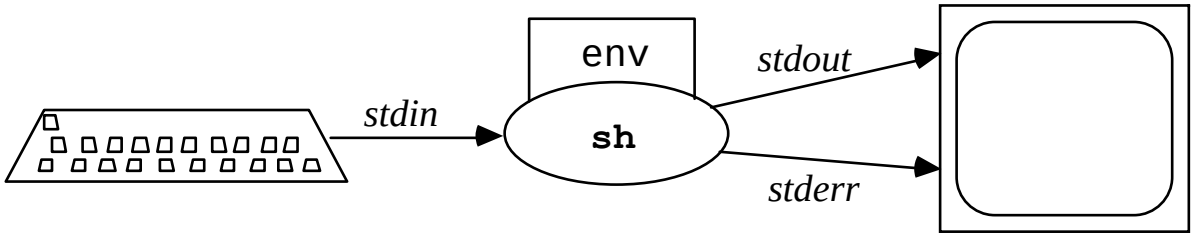
- Tout processus est créé par un autre processus
- Démarrage (*bootstrap*)
  - Création des processus de base

```
swap   PID 0
init   PID 1
```

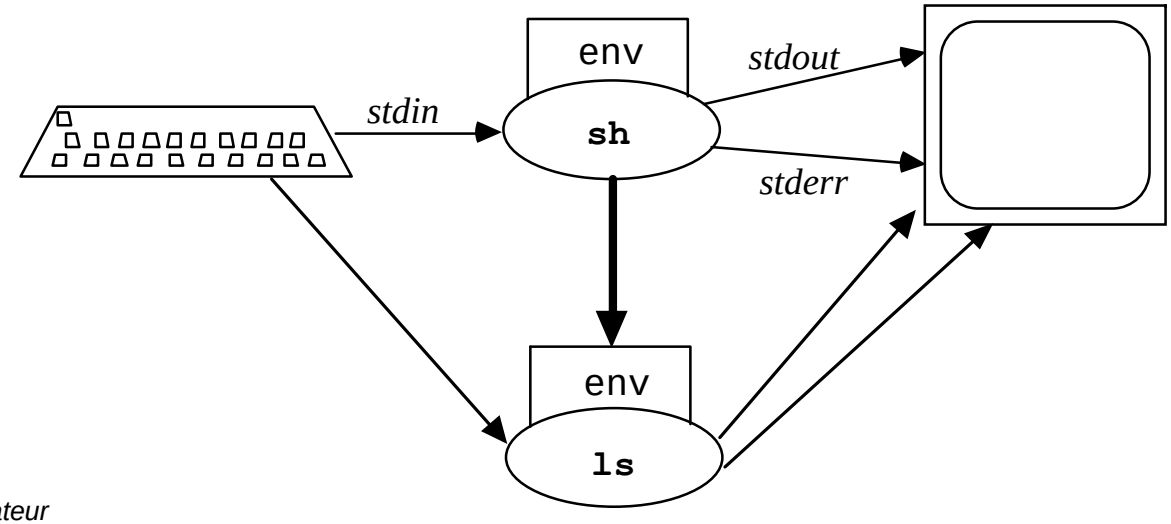


Processus : création

- Shell : père des processus utilisateur
- Environnement d'exécution



- Commande = processus fils du shell



Processus : manipulation

- Création : toute commande
- Liste des processus en cours

```
ps
ps -U <utilisateur>
```

(*process status*)

  - Autres options : voir manuel
- Lancement en arrière-plan

```
nedit tp1.c &
```

Placé derrière une commande, le caractère & indique au shell de lancer le processus mais de ne pas attendre la fin => le shell *rend la main* immédiatement à l'utilisateur.

- Arrêt d'un processus

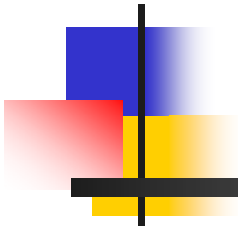
```
kill -KILL <N1> <N2> ...
```

  - Plus généralement, kill permet d'envoyer un signal à un processus

```
kill -SIG <N1> <N2> ...
kill -INT 3425
```

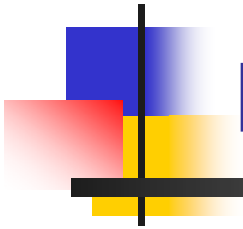
(*équivalent de Ctrl-C)*

Démo



Plan du cours

- Le SGF : système de gestion de fichiers
  - Structure arborescente
  - Utilisateur et protections
  - Commandes de base
- Les processus
  - Principe, intialisation du système
- Le langage de commande
  - Généralités
  - Environnement et variables
  - Composition des commandes
  - Écriture de scripts : paramètres, structures de contrôle
  - Fonctions et procédures



Le shell bash (Bourne again shell)

- Compatible avec le standard sh
  - Peut exécuter des scripts **`sh`**, mais l'inverse n'est pas vrai
  - C'est le shell de la *Free Software Foundation*, utilisé sur Linux
- Extensions utiles
  - Édition des commandes précédentes
  - Définition de synonymes (**`alias`**)
  - Syntaxe commode pour désigner le répertoire de travail par défaut (**`~`** et **`~utilisateur`**)

Bash : environnement

- Environnement : ensemble de variables
  - **$ env ↵** *Variables publiques*

```
_=/usr/bin/env
PATH=/usr/bin:/usr/local/bin:/users/profs/pdupont/bin:.
EDITOR=/bin/vi
LOGNAME=pdupont
MAIL=/var/mail/pdupont
SHELL=/bin/bash
HOME=/users/profs/pdupont
TERM=vt100
PWD=/users/profs/pdupont
```

  - **$ set ↵** *Variables publiques et privées*

Bash : variables (1)

- Utilisation

```
$nom            ou          ${nom}
$((nom*5))      ou          $nom*5
```

- Définition

```
NOM=valeur
Let "nom=$nom+2"
```

  - Exemple

```
PS1="Ok."
EDITOR=/bin/emacs
```

- Effacement (rarement utilisé)

```
unset nom
```

Bash : variables (2)

- Environnement initial
  - Valeurs liées à l'utilisateur : **`HOME`**, **`MAIL`**, **`SHELL`**, ...
  - Valeurs par défaut : **`PS1`**, ...
  - Valeurs définies pour tous les utilisateurs : **`/etc/profile`**
  - Valeurs définies par l'utilisateur : **`$HOME/.profile`**
- Environnements public et privé
  - Privé par défaut
  - Environnement public transmis aux sous-processus (fils)
  - Publication grâce à export

```
MANPATH=/usr/man:/usr/local/man
export MANPATH

export MANPATH=/usr/man:/usr/local/man
```

Bash : fonctionnement

- Forme générale d'une commande

```
nom [options]  paramètres ...
```

  - La plupart des commandes acceptent une liste non limitée de paramètres
- Le shell lance des sous-processus
  - Transmission de l'environnement public
  - Affectation à l'identique des flots d'E/S
- Rôle du **`PATH`**

```
/bin:/usr/bin:/usr/local/bin
```

- Modification du **`PATH`** (dans **`$HOME/.profile`**)

```
PATH=$HOME/bin:$PATH
```

Bash : composition des commandes

- Toute commande Unix retourne une valeur :

  `0` `si tout s'est déroulé normalement`

  `≠ 0` `en cas d'erreur`

  - Une commande peut être vue comme un prédicat : VRAI (0) si tout c'est bien passé et FAUX (≠ 0) en cas d'erreur
- Composition séquentielle simple (`;`)

```
$ cp f1 f2 ; mv f2 toto ↵
```

- Composition conditionnelle EtAlors (`&&`)

```
$ cc -o prog prog.c && prog ↵
```

- Composition conditionnelle OuSinon (`||`)

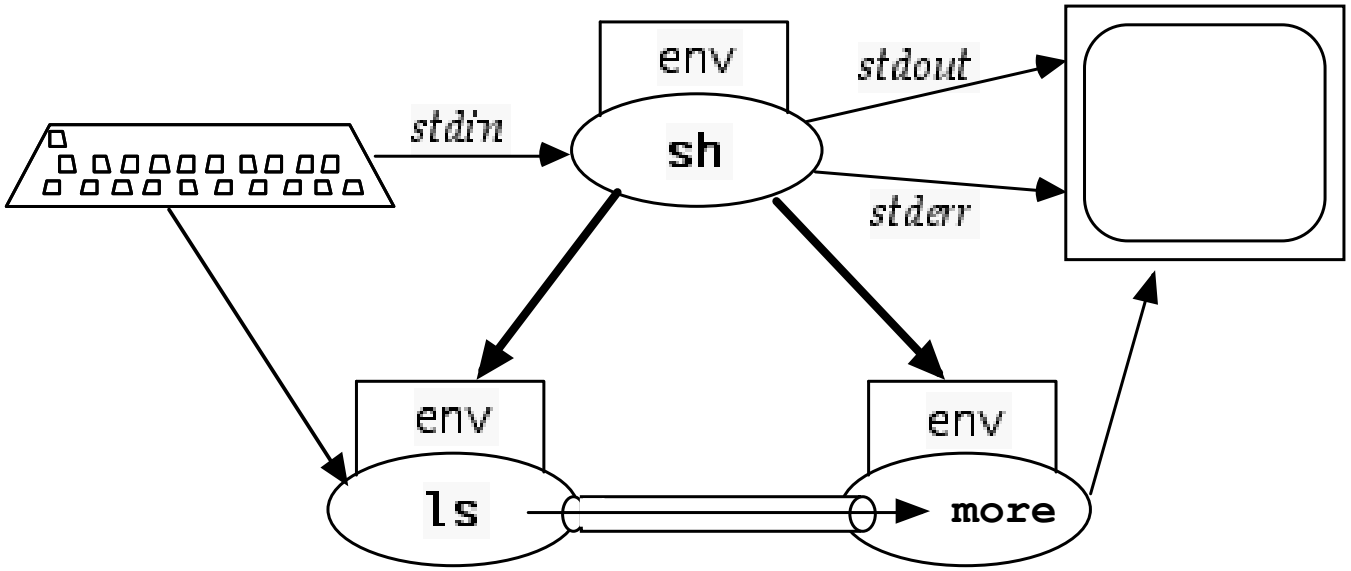
```
$ cc -o prog prog.c || echo "Erreurs !"
```

- Lancement en arrière plan (background) (&)
  - Pb : les sorties des processus d'arrière-plan apparaissent sur l'écran

Bash : tubes

- Tubes (|)
  - Redirection de la sortie d'une commande sur l'entrée d'une autre

```
$ ls -C | more ↵
```



Démo

Bash : redirection des E/S

- Sorties (**>**, **2>**, **>>**, **2>>**)
  - Flots stdout (descripteur 1) et stderr (descripteur 2)

```
$ ls *.c > liste ↵
$ cat *.c > tous_les_programmes ↵
$ cc -o tp tp.c 2> erreurs ↵
```

- Entrées (**<**, **<<**)
  - Flot stdin, descripteur de fichier n° 0

```
$ sort < atrier ↵
$ mail -s "Sujet du message" durand < message ↵
```

  - Redirection jusqu'à une certaine chaîne

```
$ mail -s Sujet martin << --FIN-- ↵
Bonjour, ↵
Ceci est le corps du message, il se ↵
termine avec la ligne suivante : ↵
--FIN-- ↵
```

Bash : caractères spéciaux, substitution

- Noms de fichiers (jokers : **`*`**, **`?`**, ...)
  - C'est le shell qui génère les listes de fichier, les commandes ne voient pas les caractères jokers
  - Essayer **`ls '*.c'`**
- Variables (**`$nom`**)

  `echo $PATH` ou **`echo $HOME`**
- Résultat d'une commande (`` ` ` `` ou **`$()`**)
  - Le résultat de la commande remplace la commande sur la ligne
  - Très utile avec des commandes comme find

```
rm `find . -name '*.o' -print`
echo `date +%d/%m/%Y`
```

- Masquage des caractères spéciaux (**`''`**, **`""`**, **`\`**)

`\c`	masque un caractère
`"<chaîne>"`	masque les caractères **`*`**,**`?`** mais pas **`$`** ni `` ` ` ``
`'<chaîne>'`	masque tout

Bash : écriture d'un script

- Script = programme shell = fichier texte
  - Création avec votre éditeur de texte favori (nedit, grasp, emacs, vi, SciTE)
- Commentaires (**#** ... ↵)

```
#Ceci est un commentaire
```

- Exécution : lancement d'un shell

```
$ bash qui ↵          ou          $ bash < qui ↵
```

  - Donner au fichier le droit d'exécution, ensuite on peut le lancer

```
$ chmod +x qui ↵
$ qui ↵
```

  - Lancement d'un sous-shell pour exécuter les commandes de qui.
- Première ligne : choix du programme d'interprétation

```
#!/bin/csh
```

- Fin de l'exécution
  - Fin du fichier ou commande **exit**

```
exit N          #N : code de retour, 0 ok, ≠0 erreur
```

Bash : entrées et sorties

- Sorties : **`echo`**
  - echo : utilise les séquences d'échappement (avec l'option -e)

```
echo Bonjour
echo -e "\nSur 3 lignes\n"
echo -e "Sur une seule\c"          pas de retour à la ligne
echo "ligne"
echo -e "\aErreur\a"
```

- Entrées : **`read`**
  - Le mot Unix : toute chaîne ne contenant ni espace, ni tabulation, ni saut de ligne (retour chariot).

```
read var1 var2 ... varN
```

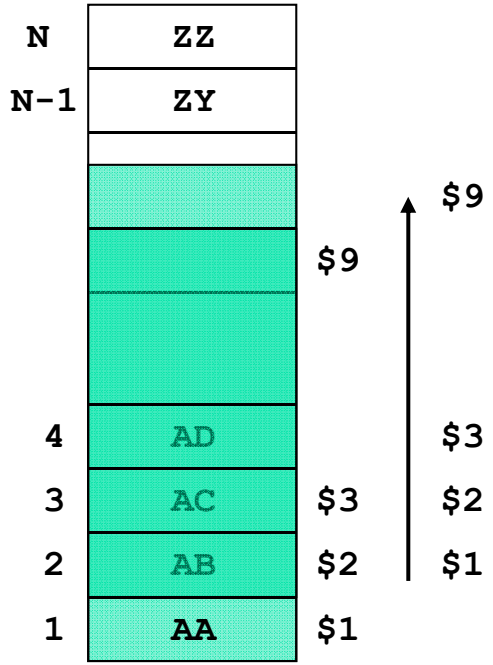
  - lit les N prochains mots et les affecte aux variables var1, var2, ...
  - Si le paramètre effectif est absent, la variable contient la chaîne vide ("")
  - S'il y a plus de paramètres effectifs que de variables, c'est la dernière (varN) qui contient les N+x derniers mots

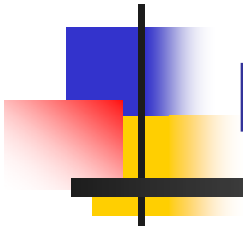
Bash : paramètres

- Paramètres (**$1**, **$2**, **. . .**, **$9**)
  - Mots Unix donnés sur la ligne de commande (chaînes de caractères)
  - Accessibles par leur position grâce aux variables spéciales **$1** à **$9**
- Accès à plus de 9 paramètres
  - Utilisation de la forme spéciale de l'instruction **for**
  - Utilisation de la commande **shift**

```
$ prog AA AB AC AD … ZY ZZ
```

```
shift
```





Bash : variables spéciales

- Quelques variables spéciales

`$#`	nombre de paramètres effectifs
`$$`	n° du processus shell en cours
`$!`	n° du dernier processus lancé en arrière-plan
`$*`	équivalent à **`"$1 $2 ... $N"`** (1 mot)
`$@`	équivalent à **`"$1" "$2" ... "$N"`** (N mots)

Bash : expressions conditionnelles

- Toute commande Unix (0 = VRAI, ≠ 0 = FAUX)
- Commande test (extrait)
  - Fichiers : FAUX si <fichier> n'existe pas

`test -r <fichier>`	VRAI si **`<fichier>`** est lisible
`test -w <fichier>`	VRAI si **`<fichier>`** peut être écrit
`test -x <fichier>`	VRAI si **`<fichier>`** est exécutable
`test -f <fichier>`	VRAI si **`<fichier>`** est normal
`test -d <fichier>`	VRAI si **`<fichier>`** est un répertoire
`test -s <fichier>`	VRAI si **`<fichier>`** a une taille > 0

  - Chaînes

`test <ch1> = <ch2>`	
`test <ch1> != <ch2>`	
`test -z <ch>`	VRAI si **`<ch> = ""`**
`test -n <ch>`	VRAI si **`<ch> != ""`**

Bash : expressions conditionnelles

- Commande test (extrait)
  - Nombres (qui sont des chaînes !)

    `test <n1> -eq <n2>` VRAI si **`<n1> = <n2>`**

    (on a aussi **`-ne`**, **`-gt`**, **`-ge`**, **`-lt`**, **`-le`**)
  - Exemple

    `test "01" = "1"` **`==>`** FAUX

    `test "01" -eq "1"` **`==>`** VRAI
  - Composition d'expressions

    `-a` (et) **`-o`** (ou)

    `!` (négation)

    `( <expr> )`
  - Facilité syntaxique

    `test <expresssion>` **`<==>`** **`[ <expression> ]`**

    `test -r tp1.c` **`<==>`** **`[ -r tp1.c ]`**

Bash : conditionnelle

```
if <commande>
then
  <instruction>
fi
```

```
if <commande>
then
  <inst1>
else
  <inst2>
fi
```

```
if <com1>
then
  <inst1>
elif <com2>; then
  <inst2>
...
else
  <instN>
fi
```

```
if test $# -eq 0
then
  echo Pas de paramètres
fi
```

```
if cc -o tp tp.c
then
  tp
else
  echo Erreurs...
fi
```

Exemples de scripts

testFic : test d'existance de fichier ou répertoire

```
#!/bin/bash
if test $# -eq 0
then
   echo Usage: testFic nomFichier
elif [ -d "$1" ]; then
   echo $1 est un repertoire
elif [ -f "$1" ]; then
   echo le fichier $1 existe
else
   echo $1 n\'existe pas
Fi
```

Affichage de la date

```
#!/bin/bash
d=$(date +%D)
mois=${d:0:2}
jour=${d:3:2}
annee=${d:6:2}
echo Nous sommes le jour $jour du mois $mois de l\'annee 20$annee
```

Bash : itérations

```
while <commande>
do
  <instructions>
done
```

```
while [ -r "$1" ]
do
  cat $1 >> liste
  shift
done
```

```
for <var> [ in <liste> ]
do
  <instructions>
done
```

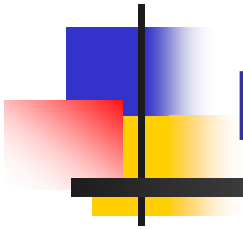
```
for f in tp1.c tp2.c
do
  cc -c $f 2> trace
done
```

```
for f in *.c
do
  cc -c $f 2> trace
done
```

```
until <commande>
do
  <instructions>
done
```

```
until [ ! -r "$1" ]
do
  cat $1 >> liste
  shift
done
```

```
echo "Paramètres :"
for f
do
  echo $f
done
```



Exemple de script

testFicMult : test de fichiers

```
#!/bin/bash
if test $# -eq 0
then
  echo Usage: testFicMult nomFichier1 ... nomFichiern
else
 for i in $*
 do
   if [ -d "$i" ]; then
          echo $i est un repertoire
   elif [ -f "$i" ]; then
          echo le fichier $i existe
   else
          echo $i n\'existe pas
   fi
 done
fi
```

Bash : branchement sélectif

- Syntaxe

```
case $<nom> in
  <sch11> [ | <sch12>... | <sch1K> ] ) <inst1> ;;
  <sch21> [ | <sch22>... | <sch2L> ] ) <inst2> ;;
  ...           ...
  <schM1> [ | <schM2>... | <schMN> ] ) <instM> ;;
esac
```

  - Les schémas sont des chaînes de caractères pouvant contenir les caractères spéciaux du shell (expressions régulières).

```
case $f in
  *.c | *.cpp )      compile -c $f ;;
  [Mn]*.h )   mv $f ../include ;;
  * )         echo "Je ne sais pas quoi faire avec $f";;
esac
```

Bash : fonctions et procédures (1)

- Regroupement de commandes
  - Dans un sous-shell : ()

```
(cd /users/2a ; ls -l)
```

  - Dans le même shell : {}
  - On peut aussi rediriger toutes les sorties

```
{ date ; ls ; who } > toto
```

- Définition d'une fonction = bloc nommé

```
nom ()
{
  instructions ...
  return valeur
}
```

```
function nom ()
{
   instructions ...
   return valeur
}
```

- Utilisation d'une fonction

```
nom param1 param2 ... paramN
```

  - Paramètres : comme les paramètres du shell, donc ceux du shell englobant ne sont plus accessibles

Bash : fonctions et procédures (2)

```
stat()
{
  if [ -d "$1" ]
  then
       echo "$1 est un répertoire"
       return 0
  else
       echo "$1 n'est pas un répertoire"
       return 1
  fi
}

stat /tmp     retourne 0
stat $dir     dépend de la valeur de $dir
stat $1       $1 ici : celui du shell principal
```

Bash : divers (1)

- Sortie brutale : exit

```
if [ $# -ne 2 ] ; then
  echo Usage : prog f1 f2; exit 1 ; fi
```

- Historique et édition des commandes précédentes
  - Navigation dans les commandes :

↑ ou Ctrl-P (previous)	commande précédente
↓ ou Ctrl-N (next)	commande suivante

  - Édition d'une commande :

← ou Ctrl-B	(backward)
→ ou Ctrl-F	(forward)
⌫ ou Ctrl-H	(delete backward)
Suppr ou Ctrl-D	(delete forward)
Ctrl-A	début de ligne
Ctrl-E	fin de ligne
!<chaîne>	relance la dernière commande commençant par **<chaîne>**

Quelques commandes utiles

- Utilisateurs
  - **`passwd`** permet le changement de mot de passe
  - **`who, w`** liste les utilisateurs connectés à la machine
  - **`write [utilisateur[@hôte]]`**

    Affiche un message sur le terminal de l'utilisateur passé en argument.
  - **`wall`** Affiche un message sur le terminal des utilisateurs connectés.
  - **`talk`** Établit une session de messagerie instantanée (chat) avec l'utilisateur passé en argument.
  - **`mesg y/n`** Autorise ou refuse les messages provenant d'autres utilisateurs.

Quelques commandes utiles

- Recherche dans des fichiers multicritères : `find`

`find <chemin(s)> <critère(s)> <action(s)>`

recherche récursive dans le(s) répertoire(s) indiqué(s) (chemin(s))

- les principaux critères (critère(s)) sont :

```
-name '<motif>'
-size <[+|-]taille>
-mtime <[+|-]date>
-user <nom|UID>
-newer <fichier référence>
```

- les principales actions (action(s)) sont :

```
-print
-ls
-exec <commande shell avec {} pour spécifier le fichier trouvé>
\;
-ok <commande shell avec {} pour spécifier le fichier trouvé> \;
```

```
$ find /home /usr -name 'ab*' -print 2> /dev/null
```

Quelques commandes utiles

- Recherche de chaînes dans des fichiers : **`grep`**

```
grep <regexp> [fichier ...]
```

Affiche uniquement les lignes, des fichiers passés en argument, correspondantes à l'expression régulière `regexp`.

`-v` inverse le résultat de la commande (affiche seulement les lignes ne correspondant pas à `regexp`)
`-c` retourne le nombre de correspondances
`-n` affiche les numéros des lignes correspondantes
`-l` affiche les noms des fichiers contenant des lignes correspondant à `regexp`
`-i` : ne tient pas compte de la casse des caractères

```
$ grep -ni "ab.." **
```

Quelques commandes utiles

- Manipulations de textes :
  - **cut**

```
cut -d<délimiteur> -f<champ(s)> [fichier]
cut -c<colonne(s)> [fichier]
```

Affiche les champs spécifiés avec l'option -f et séparés par le délimiteur indiqué après l'option -d, ou affiche les colonnes de caractères indiquées après l'option -c.

Pour afficher les 3ème et 6ème colonnes du fichier `/etc/passwd` :

```
$ cut -d":" -f3,6 /etc/passwd
$ cut -c1-10 /etc/passwd
```

Quelques commandes utiles

- Manipulations de textes :
  - **wc** `wc <fichier ...>`

Affiche le nombre de lignes, de mots et de caractères (*Word Count*) contenus dans les fichiers passés en arguments.
`-l` affiche uniquement le nombre de lignes (line)
`-w` affiche uniquement le nombre de mots (word)
`-c` affiche uniquement le nombre de caractères (character)

```
$ wc /etc/services
   569 2805 19935 /etc/services

$ ls | wc -l
```

Quelques commandes utiles

- Tri de fichiers: **`sort`**, **`uniq`**
  - `sort` trie le fichier en ordre croissant
  - `uniq` permet d'éliminer les lignes en double ou de compter les doubles

```
$ sort namesd.txt | uniq
$ sort –u namesd.txt
$ sort namesd.txt | uniq –c
      2 Alex Jason:200:Sales
      2 Emma Thomas:100:Marketing
      1 Madison Randy:300:Product Development
      1 Nisha Singh:500:Sales
      1 Sanjay Gupta:400:Support
```

- Information sur le type des fichiers : **`file`**
- Outils plus complets : **`sed`**, **`awk`**, **`perl`**