Systèmes d'exploitations : usage et principes

Parce que *"Systèmes communiquant en réseau"*, ça ne veut pas dire grand-chose...

30 novembre 2021

iut
Université d'Auvergne

Plan

Conventions d'écriture

- Une *notion importante* nouvelle
- **En gras**, un point particulier
- Une `commande`, ou un `nom technique`.
- `$` ou ` ` en début de ligne désigne le prompt avant une commande.
- `f1` désigne le nom d'un fichier, `d1` est le nom d'un répertoire.

Système d'exploitation

- Un élément *logiciel*
- Masque le **matériel** (processeur, mémoire, disques durs/SSD…)
  - Permet aux *processus* (programmes en cours d'exécution) d'y accéder via des *appels systèmes* exécutés par le *noyau*
  - Permet aux *utilisateurs* d'y accéder via des *commandes*

Un dessin ...

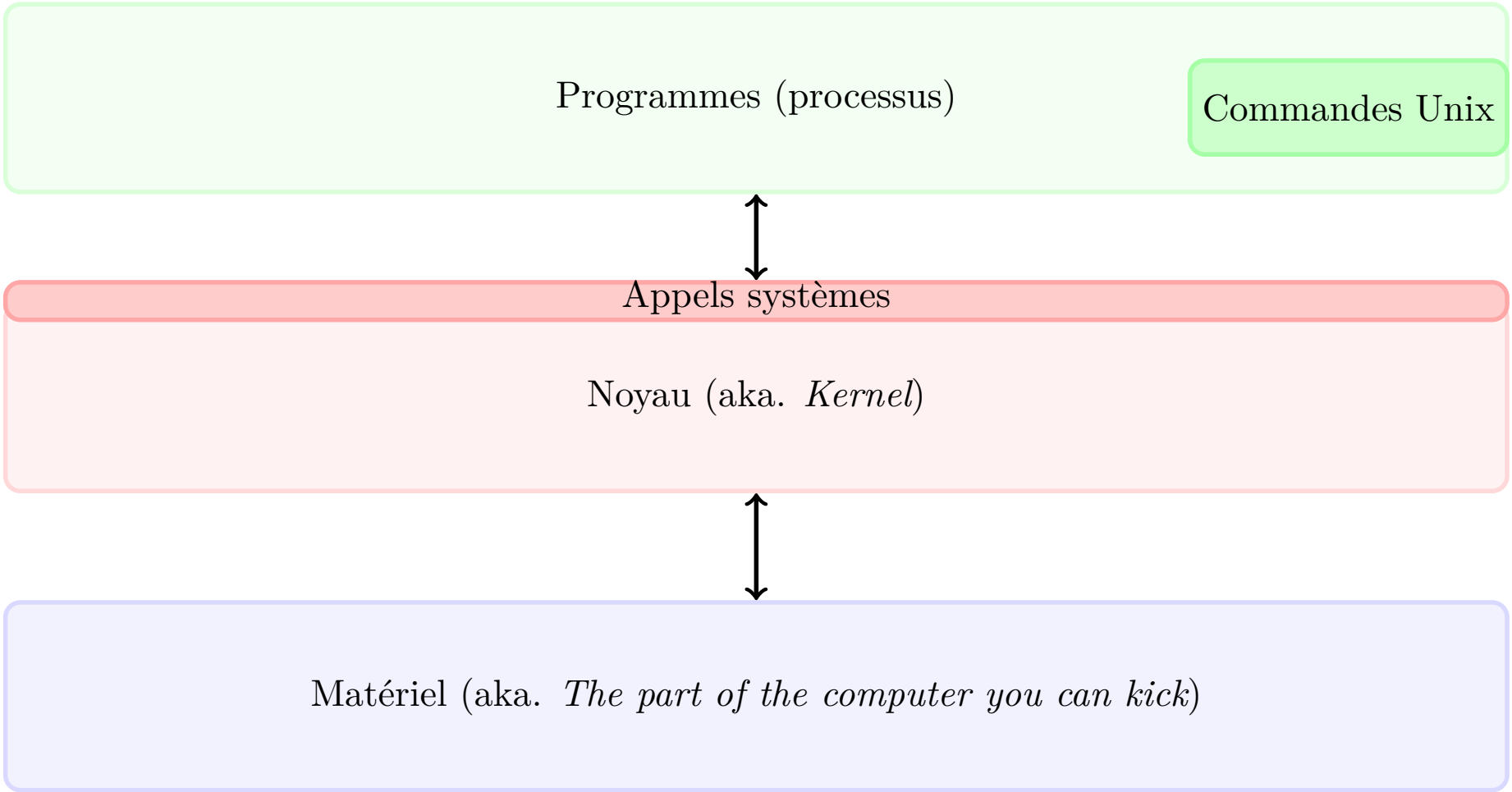


Figure – Système d'exploitation <-> Noyau + Commandes

Objet de ce cours

- Apprendre les *commandes Unix* (Linux est un Unix parmi d'autres) permettant à l'utilisateur de contrôler le système d'exploitation (SE).
- Écrire des *scripts* faits de commandes pour manipuler les ressources du SE.
- Comprendre les principes de bases du système d'exploitation.

Fichiers

- L'abstraction qui nous masque le disque dur/SSD, mais aussi la plupart des périphériques
- **Mantra** : *En Unix, tout est fichier* :
  - Fichier simple : contient des données
  - Répertoire : fichier qui contient d'autres fichiers
  - Un *ddur* (disque dur, ou SSD) est un fichier (`/dev/sda`), une *partition* est un fichier (`/dev/sdb2`)
  - Un *lien symbolique* (comme un raccourci) est un fichier.

Disque dur, partitions, systèmes de fichiers

- Un ddur permet de stocker des octets en vrac.
- Il est découpé en *partitions*. Les partitions sont énumérées dans une *table des partitions*.
- Sur chaque partition, on écrit un *système de fichiers* (ex : `ext4`, `zfs`, `NTFS`)
- Les systèmes de fichier contiennent des fichiers (au sens Unix du terme)

Types de systèmes de fichiers

- Usage *local* à la machine (Disque dur / SSD, média amovibles) : `ext4`, `zfs`, `ntfs`, `fat`.
- Usage *distant* (réseau), comme `NFS (v4)`.
- Usage en RAM (mémoire vive), comme `tmpfs`.

Organisation de fichiers

- Organisés en *arborescence*.
  - Tout nœud a un père unique
  - Sauf la *racine* (le nœud le plus haut)
- Chaque partition est *montée* sur un nœud de l'arbre.
- `mount` permet de voir la liste des systèmes de fichiers montés
- `df -h` aussi, avec des infos sur l'espace utilisé

Se repérer dans l'arborescence : chemin absolu

- Pour désigner un nœud dans l'arbre, on donne un *chemin*
- Un *chemin absolu* part de la *racine* nommée `/`
  - On note ensuite tous les répertoires traversés, séparés par des `/`

`/home/delobel/tmp/2021-01-05_10-28.png`

- On part de la racine (`/` qui contient un répertoire `home`). On traverse `/home` pour atteindre `/home/delobel` ...

Se repérer dans l'arborescence : chemin relatif

- Tout processus (dont le `shell`) a un *répertoire courant* (celui dans lequel on se trouve)
  - **Commande** : `pwd` indique le répertoire courant du shell
- Un *chemin relatif* est le chemin qui part du *répertoire courant* jusqu'à la destination
- Si je suis déjà dans `/home/delobel`, le chemin *relatif* du fichier exemple précédent est `tmp/2021-01-05_10-28.png`
- Si je suis dans `/home`, chemin relatif est `delobel/tmp/2021-01-05_10-28.png`

Quizz

Est-ce un chemin correct ? Absolu ou relatif ?

- `/etc/passwd`
- `/etc/`
- `/etc`
- `fillon/rend/l_argent`
- `fillon`
- `/tmp/2021-01-05_10-28.png`

Chemins spéciaux

- `$HOME` : chemin absolu désignant votre répertoire de connexion. Aussi appelé *racine de votre compte*. Raccourci : `~`.
- `..` répertoire parent (niveau d'au-dessus)
- `.` répertoire courant

Commandes

- **Toujours** la *commande* en *premier* sur la ligne
- Les autres informations (*arguments*) sont séparés par des espaces (une ou plusieurs)
- `-p` est une *option* de `mkdir` (en forme courte)
- `--parents` est la même option, en *forme longue*
- Une option est propre à une commande.
- **Mantra** : *Pas de nouvelles, bonne nouvelle !*
  - Si tout va bien, la commande est la plupart du temps silencieuse.
  - Si ça merde, un *message d'erreur* est affiché. **On le lit, on le comprend** !

Comment modifier le répertoire courant ?

- **Commande** : `cd` pour *Change Directory*.
  - `cd chemin` : change le répertoire courant à `chemin`
- **Pratique** : `cd` (sans argument) vous ramène à votre répertoire de connexion
- Quizz. Que font les commandes suivantes ?
  - `cd ..`
  - `cd /`
  - `cd ~`
  - `cd /; cd etc`
  - `cd .`

Savoir quel est le répertoire courant actuel

- **Commande** : `pwd` pour *Print Working Directory*
  - `pwd` : affiche le chemin absolu du répertoire courant

Regarder l'arborescence en dessous

- **Commande** : `tree`

```
.
|-- d2
|   |-- f2.1
|   `-- f2.2
|-- d3
`-- f1
```

- **Option utile** `-d` pour afficher seulement les répertoires

```
tree -d
```

```
.
|-- d2
`-- d3
```

Créer un répertoire

- **Commande** : `mkdir` pour *MaKe DIRectory*
- `mkdir rep1`
- `mkdir d1 d2`
- `mkdir r1; cd r1; mkdir r2`
- `mkdir r1; mkdir r1/r2`
- `mkdir -p r1/r2`

`man`, pour *Manuel*

- `man mkdir`
- **Mantra** : *Le man est votre ami*
- Moins d'informations : `mkdir --help`
- Utile aussi pour le C (chapitres 3 et 2) : `man 3 printf`

Voir le contenu d'un répertoire

- `ls chemin` : affiche le contenu de chemin si c'est un répertoire, juste le nom sinon.
- `ls` vaut `ls .` : par défaut, `chemin` vaut `.`
- Affichage du détail avec `-l`
- `ls -l /etc` ?
- `ls -l /etc/passwd`
- Plus de 50 options disponibles ! *Takafairman* !

Fichiers cachés

- Tout fichier dont le nom commence par un `.` est *caché* (il n'apparaît pas dans un `ls` normal)
- **Commande** : `touch cheminfichier` permet de créer un fichier

```
$ touch /tmp/d1/f1
$ ls /tmp/d1
d2  d3  f1
$ touch /tmp/d1/.cache
$ ls /tmp/d1
d2  d3  f1
$ ls /tmp/d1/.cache
/tmp/d1/.cache
$ ls -a /tmp/d1
.  ..  .cache  d2  d3  f1
```

- L'option `-a` (All) de `ls` permet d'afficher aussi les fichiers cachés

Copier des fichiers

- `cp` pour *CoPy*. `cp source destination`
  - `source` et `destination` sont des *chemins*
- `cp f1 f2` : Que se passe-t-il si `f2` existait avant ?
- Multiples fichiers : `cp f1 f2 f3 /tmp`
- `cp f1 f2 f3` : Quelle est l'erreur ?

Supprimer un répertoire vide

- `rmdir` pour *ReMove DIRectory* : `rmdir d1`
- Ne fonctionne que si le répertoire ne contient rien (même pas de fichiers cachés)

Supprimer un fichier

- `rm` pour *ReMove*. `rm chemin`
- Multiples fichiers et répertoires : `rm f1 f2 f3`
- Supprimer des répertoires : besoin de l'*option* `-r` (Pour *Recursive*)
  - `rm -r d1 f1`
- Que fait `rm -r ~` ?

Jocker : `*`

- Caractères remplacés par le shell *avant* l'exécution de la commande.
- `*` : n'importe quels caractères, autant que l'on veut `ls`

```
1 f1.pdf f2.pdf f1.txt d1
```

- Examples :
  - `cp *.pdf /tmp`
  - `rm *.pdf`
  - `mv /tmp/*.pdf d1`
  - `rm /tmp/f*.txt`

Montage des partitions

Une *partition* est dite *montée* quand elle est associée à l'*arborescence* pour qu'on puisse utiliser le *système de fichiers* qu'elle contient.

Exemple de partitions montées

```
Sys. de fichiers              Taille Utilisé Dispo Uti% Monté sur
dev                             16G       0   16G   0% /dev
run                             16G    1,6M   16G   1% /run
/dev/nvme0n1p2                 196G    106G   81G  57% /
tmpfs                           16G    114M   16G   1% /dev/shm
/dev/nvme0n1p1                 500M     35M  465M   7% /efi
/dev/nvme1n1p1                 458G    210G  225G  49% /mnt/steam
/dev/sda2                       23G    9,0G   13G  43% /var
tmpfs                           16G    4,3M   16G   1% /tmp
/dev/sdb2                      586G    178G  379G  32% /mnt/attic
tmpfs                          3,2G      60K  3,2G   1% /run/user/1000
troll:/mnt/toblerone/coffre    1,8T    182G  1,7T  10% /mnt/troll/coffre
troll:/mnt/films               916G    687G  183G  79% /mnt/troll/films
/dev/sdc1                      1,8T    941G  800G  55% /mnt/suisse
/dev/sdd1                      2,7T    1,8T  851G  68% /mnt/grenier3T
/dev/sdf1                       15G      24K   14G   1% /run/media/delobel/usb3-16G
```

Pour comprendre la sortie précédente

- `/dev/sda2` : Seconde partition (`2`) du premier disque (`a`). `sd` : Bus disques classiques (SATA...), monté sur le répertoire de chemin `/var` .
- `/dev/nvme0n1p2` : Premier Disque M.2 PCI Express (`nvme0`), 2ème partition (`2`).
- Le répertoire `/tmp` est un disque en mémoire vive (RAM), car il est de type `tmpfs` .
- `troll:/mnt/toblerone/coffre` : disque réseau, venant de la machine `troll` , exportant le chemin `/mnt/toblerone/coffre` , monté sur le répertoire `/mnt/troll/coffre` .
- `dev` : système de fichiers qui permet au noyau d'exporter les périphériques (DEVices).
- `run` : comme `tmpfs` , mais dédié aux processus en cours.

Comment avoir plus d'information ?

- On fait un `mount` en récupérant seulement les infos sur `sda2` : `mount | grep sda2`

```
/dev/sda2 on /var type ext4 (rw,relatime)
```

- Le système de fichiers de `sda2` est `ext4`

`mount | grep coffre` :

```
troll:/mnt/toblerone/coffre on /mnt/troll/coffre type nfs4
(rw,relatime,vers=4.2,rsize=1048576,wsize=1048576,namlen=255,
timeo=600,retrans=2,sec=sys,clientaddr=192.168.2.4,local_lock
```

- `nfs4` : système de fichiers `NFS`, version 4.
- Adresse IP de troll (le serveur) : `192.168.2.111`
- Adresse IP du client : `192.168.2.4`
- Monté en lecture / écriture `rw` (Read and Write).

Questions

- Quels types de fichiers différents connaissez vous sous Unix ? Type de fichier, pas type de contenu.
- Que font les options de `ls` suivantes : `-l`, `-d`, `-lh`, `-lrt` ?
- Où nous amènent :
  - `cd ~`
  - `cd ..`
  - `cd .`
  - `cd /`
  - `cd d1`
  - `cd ../d1`
  - `cd $HOME`

Flux d'entrée/sortie (IO) standard

Un *processus* est un programme en cours d'exécution.
Tout *processus* possède 3 flux (stream) :

- Entrée standard : **0** (*stdin*). Défaut : clavier.
- Sortie standard : **1** (*stdout*). Défaut : écran.
- Sortie d'erreur : **2** (*stderr*). Défaut : écran.

Ces flux sont *hérités* du processus père.
Afficher en shell, c'est mettre sur la sortie standard.

Redirection de flux en Shell vers des fichiers

- `> fichier` : Redirige *sortie standard* vers *fichier* (avec écrasement du fichier).
- `>> fichier` : Redirige *sortie standard* vers *fichier* (ajout en fin de fichier).
- `< fichier` : Redirige *entrée standard* depuis *fichier*.
- `2> fichier` : Redirige *sortie d'erreur* vers *fichier*.
- `&> fichier` : Redirige *sortie standard* **et** *sortie d'erreur* vers *fichier*.
- `>&2` : Redirige *sortie standard* vers *sortie d'erreur*.

Écrasement : le contenu précédent du fichier est effacé.

Quelques fichiers spéciaux

- `/dev/null` : Un fichier “poubelle” qui détruit tout ce qu’on met dedans !
- `/dev/zero` : Ce fichier contient une infinité d’octets à 0.
- `/dev/random` : une infinité d’octets aléatoires.

`ls /home/etud/* 2>/dev/null` : On ignore les erreurs !

Redirection de flux en Shell vers des fichiers

- `date > date.txt`
- `sort < fic.txt > fic.trié.txt`
- `ls *.c 2> /dev/null`
- `echo "Erreur !" >&2`

Les tubes (pipes)

- Redirige la *sortie standard* d'un processus vers *l'entrée standard* d'un autre processus.
- Les commandes sont exécutées en *parallèle* (en même temps)
  - Elles sont *synchronisées* sur les données du tube.

```
commande1 arg1 | commande2 | ... | commandeN
```

- `Broken Pipe` : Le *lecteur* est mort avant l'*écrivain*.

`wc` et `cat`

- `wc` : Word Count. Compte les mots, les caractères, les lignes.
  - `wc fichier` : Compte le contenu de `fichier`
  - `wc` : Attend frappe clavier (entrée standard)
    - → On indique la fin par `Ctrl + d` → compte ce qui a été tapé
  - `wc < fichier` : Compte sur l'entrée standard, donc le contenu de `fichier`
  - `-l` : compte les *Lignes*
  - `-c` : compte les *octets*
  - `-w` : compte les *Words* (mots)
- `cat f1 f2...` : met le contenu de `f1` puis `f2` sur la sortie standard
  - Par défaut, affiche
- `cat f1 f2 | wc` : Compte le contenu de `f1 + f1`.

`/etc/passwd` : Un fichier texte présent partout

1 login : 2 password : 3 uid : 4 gid : 5 commentaire : 6 home : 7 shell

```
1  root:x:0:0:root:/root:/bin/bash
2  delobel:x:1000:1000:Francois Delobel,,,:/home/delobel:/bin/zsh
3  daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
4  bin:x:2:2:bin:/bin:/usr/sbin/nologin
5  sys:x:3:3:sys:/dev:/usr/sbin/nologin
6  sync:x:4:65534:sync:/bin:/bin/sync
7  man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
8  lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
9  mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
0  www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
1  sshd:x:110:65534::/var/run/sshd:/usr/sbin/nologin
2  ...
```

- UID : User ID, nombre identifiant un utilisateur.
- GID : Group ID, nombre identifiant un groupe.
- home : répertoire de connexion.

Quizz

Que font les commandes suivantes ?

- `ls | wc -l`
- `cat /etc/passwd | wc -l`
- `wc -l < /etc/passwd`
- `wc -l < /etc/passwd > f1`
- `ls | wc -l > f1 2> f2`

`grep` sélectionne des lignes

- `grep motif` filtre les lignes suivant leur contenu.
  - Si une ligne contient le *motif* elle est mise sur la sortie standard.
  - Sinon, elle est ignorée (donc jetée).
- Si `grep motif f1 f2...` : cherche dans le contenu des fichiers.
  - Si `grep motif`, cherche dans l'entrée standard.
- `cat /etc/passwd | grep /bin/bash` : garde les lignes de `/etc/passwd` contenant la chaîne `/bin/bash`
  - `cat /etc/passwd | grep /bin/bash | wc -l` : ?

`cut` découpe des lignes

- `-d caractere` : le caractère Délimiteur de champs
- `-f numero` : Le numéro du champ (Field) à conserver.
  - Tout le reste est perdu
- `cat /etc/passwd | cut -d ':' -f 1` : Affiche le premier champ du fichier `/etc/passwd` (user)
- `numero` peut être complexe
  - `1-4` : les quatre premiers champs.
  - `1,4` : le premier et le quatrième champ.
  - `3-` : le troisième champ et les suivants.
  - `1,3-5,7` : ?

`sort` pour trier, `uniq`

- `sort` : trie les lignes (alphabétique).
  - `-n` : trie des chiffres (Numeric)
  - `-r` : ordre inverse (Reverse)
  - `-R` : Random (mélange)
  - `-f` : ne tient pas compte de la casse (minuscules/majuscules)
  - `cat /etc/passwd | sort -n -k 3 -t ':'` : Trié selon les `uid` (champ 3)
- `uniq` : supprime les doublons consécutifs.
  - `-c` : affiche le nombre d'occurences.

Quizz

```
1 cat /etc/passwd | cut -d ':' -f 1 | sort > /tmp/log.txt

2 cat /etc/passwd | cut -d ':' -f 7 | sort | uniq -c

3 cat /etc/passwd | sort -n -k 3 -t ':' | cut -d ':' -f 1 > /tmp/log.txt
```

tr

- Pour TRanslate : modifications de caractères
- `-d chars` : Détruit les caractères
  - `echo 'Les mots sont importants' | tr -d oa`
    - → `Les mts snt imprtnts`
- `tr from to` : Remplacer un caractère par un autre
  - `echo 'Les mots sont importants' | tr oa '-'`
    - → `Les m-ts s-nt imp-rt-nts`
  - `echo 'Les mots sont importants' | tr oa '-?'`
    - → `Les m-ts s-nt imp-rt?nts`
- `tr -s` : supprime les répétitions (Squeeze)
  - `echo 'aaaaabbbbbccccccaaaa' | tr -s 'ab'`
    - → `abccccca`

Example de mélange `tr` et `cut`

```
$ ls -l
-rw-r--r--  1 delobel delobel  21866 22 mars  17:23  RESULTATS_C++.ods
drwxr-xr-x 73 delobel delobel   4096 19 juil. 10:40  Sailfish
lrwxrwxrwx  1 delobel delobel     34 25 août    2016  Seafile -> /mnt/greni
```

Des espaces sont ajoutées pour l'alignement ⇒ cut pas possible !

```
$ ls -s | tr -s ' '

-rw-r--r-- 1 delobel delobel 21866 22 mars 17:23 RESULTATS_C++.ods
drwxr-xr-x 73 delobel delobel 4096 19 juil. 10:40 Sailfish
lrwxrwxrwx 1 delobel delobel 34 25 août 2016 Seafile -> /mnt/grenier/homes
```

```
$ ls -s | tr -s ' ' | cut -d ' ' -f 5,9

21866 RESULTATS_C++.ods
4096 Sailfish
34 Seafile
```

Autre commandes

- `head -n 10` : affiche les 10 premières lignes.
- `tail -n 10` : affiche les 10 dernières lignes.
- `more` : affiche page par page.
  - `/` : recherche dans more.
  - `q` : quitter.
- `less` : fait pareil que `more` en mieux.
- `echo` : affiche.
- `printf` : affiche avec formatage.

Jokers

- Interprétés par le *shell* **avant** l'exécution de la commande.
- Deux cas
  - En première position de la ligne de commande : cherche une *commande* dans le `PATH`
  - Après : cherche un *fichier* (chemin)
- `echo *` : `ls` du pauvre

Joker : `*`

- `*` : n'importe quelle chaîne de caractère (même vide).
  - **Exception** : *suivant la configuration du shell* en début de mot, ne peut commencer par `.` (fichiers cachés)
  - Dans ce cas, utiliser aussi `.*`
- `rm *.o`
- `cp f*.pdf /tmp`

Joker : ?

- `?` : **exactement** un caractère (n'importe lequel).
- `echo ????` : affiche le nom des fichiers de 4 caractères.
- `echo ????*` : affiche le nom des fichiers de 4 caractères **ou plus**.
- `rm *.???` : Si fichiers : `troll`, `lapin.pdf`, `connect.c`, `un.drole.de.fichier.txt`

Joker : `[]`

- `[]` : **exactement** un caractère parmi ceux entre crochets
- `[aeiouy]` : une voyelle
- `[lapin]` : une lettre parmi `a`, `i`, `l`, `n`, `p`
- `[a-z]` : une minuscule
- `[0-9]` : un chiffre
- `[!0-9]` : tous **sauf** un chiffre
- `[0-9+-]` : un chiffre, un `+`, ou un `-`

Jocker : `[[:classe:]]`

- `[[:alpha:]]` : une lettre
- `[[:upper:]]` : une majuscule
- `[[:space:]]` : un séparateur blanc (espace, tab…)
- alnum, alpha, ascii, blank, cntrl, digit, graph, lower, print, punct, space, upper, word, xdigit
- `man bash` : chercher *“character classes”*.

`vi`, éditeur de texte en texte

- `vi` → `vim` (Vi IMproved).
- Présent sur tout système Unix.
- Léger en réseau (distant), CPU.
- Super customisable mais compliqué à customiser.

`vim` : 2 modes

- *Mode édition* : les touches font du texte.
- *Mode commande* : les touches sont des commandes.
- Quitter le mode d'édition : `ESC` (revient en mode commande)
- Entrer en mode d'édition : `i` (insert), `O` : ajoute une ligne avant, `A` : à la fin
- Usage : `vim myscript.sh`

`vim` : quelques commandes de base

- `/aiguille + RET` : cherche le mot *aiguille*
  - `n` : Next (occurrence suivante)
  - `Shift + n` : occurrence précédente.
-
  - `dd` : supprime une ligne.
  - `7dd` : détruit 7 lignes
- `p` : paste (colle) ce qui a été copié / supprimé.
- `u` : undo.
- `:w` : écrit le fichier.
- `:q` : quitte
  - `:wq` : ? (ou `ZZ`)
  - `:q!` : Quitter sans sauvegarder !
- `gg=G` : indenter tout le fichier (suivant le type de fichier)

`vim` : copier coller

- **Attention** : pour *coller* dans un terminal, faites un `Ctrl + Shift + v`. (Copier : `Ctrl + Shift + c`).
- Mode sélection de vim (Visual) : `v`
  - Sortie par `Esc`.
  - Sélection en se déplaçant.
  - `y` pour copier.
  - `x` pour couper.
  - `p` pour coller.
- Variantes du mode de sélection :
  - `V` : sélection par *lignes* (V-LINE).
  - `Ctrl + v` : sélection par bloc rectangulaire (V-Block)

Modifier la config de `vim`

- `cp /etc/vim/vimrc ~/.vimrc; vim ~/.vimrc` : éditez votre fichier de configuration.
- `" commentaire` : Original, non ?
- `syntax on` : activer la coloration syntaxique par défaut.
- `filetype ...` : prise en compte des types de fichier.
- `colorscheme solarized` : choix du thème.
  - Pour tester les différents thèmes, ouvrez un fichier et `colorscheme TAB` pour défiler les thèmes

Pour aller plus loin

Pour les passionnés (sisi, y'en a !)

- `vimtutor` : tutoriel interactif de vim
- `neovim` : réécriture de `vim` pour plus de vitesse

Pourquoi des droits ?

- Unix est né pour des machines partagées et est majoritaire pour les serveurs (100% du top 500 des supercalculateurs, 77% des 10 millions de sites les plus consultés). On doit gérer des *utilisateurs* différents.
- Pour des raisons de *sécurité*, les services (serveur web par exemple) sont associés à des utilisateurs différents.
- Un système d'exploitation gère donc des *permissions* sur les *ressources* (fichiers, processus)

Droits sur les fichiers

```
ls -ld /etc/passwd /etc/shadow /home/*
```

```
1 drwxr-xr-x  10 ivan    ivan     4096 22 déc.   2019 /home/ivan
2 drwx--x--x 116 delobel delobel 12288 20 août  17:17 /home/delobel
3 -rw-r--r--   1 root    root     2275  4 avril 23:00 /etc/passwd
4 -rw-------   1 root    root     1261  4 avril 23:00 /etc/shadow
```

- Le premier caractère indique le *type de fichier* : `d` pour répertoire (Directory), `-` pour *fichier simple*
- Les 9 caractères suivant sont trois groupes de droits (UGO)
  - Pour le *propriétaire* du fichier (User)
  - Pour le *groupe* du fichier (Group)
  - Pour les autres (Others)

Détail des droits

- `rwx` : Lecture (`r`), Écriture (`w`), Exécution (`x`)
- `-rw-r--r-- root root /etc/passwd` : User ? Group ? Other ?
- `-rw------- root root /etc/shadow` : User ? Group ? Other ?
- `drwxr-xr-x ivan ivan /home/ivan` : User ? Group ? Other ?
- `drwx--x--x delobel delobel /home/delobel` : User ? Group ? Other ?

Droits sur les fichiers **simples**

- Lecture (Read) : Lire le *contenu* du fichier
  - Permet la copie par exemple, ou un `cat`
- Écriture (Write) : Modifier le *contenu* du fichier
  - Permet l'édition
- Exécution (eXecute) :
  - Permet de créer un *processus* qui exécute le programme
  - `-rwxr-xr-x root root /bin/ls`

Droits sur un **répertoire**

- Lecture (Read) : obtenir la liste de ses fichiers
  - `ls /tmp/d1`
- Écriture (Write) : créer/supprimer des fichiers
  - `touch /tmp/d1/f4`
  - `rm /tmp/d1/f4`
- Exécution (eXecute) : accéder aux fichiers (ou *traverser* le répertoire)
  - `cd /tmp/d1`
  - `cat /tmp/d1/f1`

Modification des droits

- **Commande** : `chmod` (CHange MODe)
  - `chmod u=rw,g=r,o= /tmp/d1/f`
  - `=` : Fixe les droits pour `u`, `g`, ou `o`
  - `chmod go+x /tmp/d1`
  - `+` : ajoute un droit aux droits existants
  - `chmod g-r /tmp/d1/f1`
  - `-` : retire un droit aux droits existants
  - `chmod ug+r,o= /tmp/d1/f1`

`chmod` en numérique

- 4 : `r`
- 2 : `w`
- 1 : `x`
- `rx` : 4 + 1 : 5
- `rw- rw- ---` → `u:rw, g:rw, o:`
  - (6)(6)(0) → 660
- `chmod 750 /tmp/d1` ?

Droits étendus

- Les droits de bases *Unix* ne suffisent pas dans certaines situations.
- Linux ajoute les *ACL* (Access Control Lists)
- Pas vus dans ce cours (voir `getfacl`, `setfacl`)

Questions

- Quels droits donne un `chmod 640` ?
- Comment rendre un fichier `f` exécutable ?
- Quels sont les droits du fichier `/bin/chmod` ?

Trouver en parcourant : `find`

- **Commande** `find` : parcourt l'arborescence, effectue des actions. Par défaut, affiche (d'où le nom).
- Exemples :
  - `find ~ -iname '*.c'` : Parcourt votre compte, vérifie que le nom du fichier correspond à `*.c`, affiche (par défaut).
  - `find /home -iname '*.c'` : même chose pour le compte de *tous* les utilisateurs.
    - → Comment ne pas voir les erreurs ?
- Mettre plusieurs filtres (avec des *et* entre eux)
  - `find rep -filtre1 -filtre2 -filtre3`
  - Affiche les fichiers de l'arborescence `rep` qui vérifient *filtre1* **et** *filtre2* **et** *filtre3*.

`find` : filtres

- Évalués de gauche à droite : “And else”
- `-size n\[bckw\]`
- `-empty`
- `-iname pattern`
- `-type` `d` pour Directory, `f` pour File (fichier simple), `l` pour lien (raccourci)
- Chemin : `find /usr -path '*/linux-h*'`
- Éviter une branche : `find / -path '/dev' -prune -o -print`
- Plein d’autres -> `man find`

`find` et critères numériques

- `42` : exactement 42
- `+42` : Plus de 42
- `-42` : Moins de 42
- `find /etc -size -30 -size +10`

Parfois des unités

- `find -iname '*.pdf' -size +10M`

`find` et temps

- `m` : Modification time (Modification contenu)
- `c` : `chomd` time (metadata)
- `a` : Access Time (Pas forcément stocké)
- `min` : minutes (`-30` : moins de 30mn)
- `time` : 24h (`-2` : moins de 48h)
- `newer` : rapport à la date d'un autre fichier
- `find /etc -mtime -2`
- `find ~ -mmin -120 -mmin +30`
- `find /etc -newer /etc/hosts`

`find` : Actions

- Action *par défaut* : `-print` (affiche)
- `-delete` : détruit
  - `find ~ -iname '*.o' -delete`
- `-ls` : `ls -l`
- `-printf` : comme en **C**.

Quizz

Équivalents ?

- `find ~ -iname 'f*' -print -iname'*.o'  -delete`
- `find ~ -iname 'f*' -print -delete -iname '*.o'`
- `find ~ -delete -iname 'f*' -print -iname '*.o'`

`find` et `-exec`

- `-exec` permet d'exécuter n'importe quelle commande !
  - `{}` : nom du fichier
  - `\;` : fin de la commande
- `find /usr/share -iname '*.pdf' -exec cp {} /tmp \;`
- `find $HOME -iname '*.pdf' -size +10M -exec gzip {} \;`

Questions

- Comment supprimer tous les fichiers `.o` de votre compte ?
- Comment lister les fichiers de votre arborescence qui ont été modifiés depuis une heure ? Avec le détail ?
- Comment rechercher un fichier PDF dont le nom contient ‘password’ et dont la taille dépasse 5MiB ?

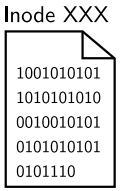
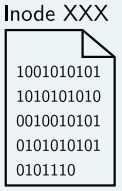
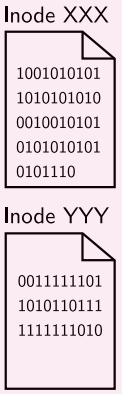
Les liens (1/2)

- Deux types :
  1. Lien **physique** (*hard link*) : deuxième nom pour un même fichier. Pas vu dans ce cours
  2. Lien **symbolique** (*symbolic link*, *symlink*) : fichier qui contient le chemin d'un autre fichier (raccourcis).
- `ln -s` crée un *lien symbolique*.
  - `ln -s /etc/password ~/pass` : Crée un fichier `~/pass` qui pointe vers `/etc/password`
- Pour les autres opérations, utilisez : `rm`, `mv`, `cp`, ...
- Par défaut `find` ne déréférence pas les liens symboliques (cf. `-follow` `-noleaf`)

Attention

N'oubliez pas l'option `-s` : `ln -s filename linkname`

Les liens (2/2)

Action	Commande	Représentation abstraite	Représentation physique
Création d'un fichier	`touch myFile.dat`	`myFile.dat` (fichier original)	
Création d'un lien physique	`ln myFile.dat otherName`	`myFile.dat` (fichier original) `otherName` (lien dur)	
Création d'un lien symbolique	`ln -s myFile.dat linkName`	`myFile.dat` (fichier original) `linkName` (lien symbolique)	

Les processus : généralités

Processus

- Processus <=> programme qui s'exécute.

Deux parties :

1. un *code exécutable* (le programme) ; suite d'instructions en langage machine, éventuellement partageable entre les processus
2. un *contexte d'exécution* propre à chaque processus

Les processus : l'arborescence

Tout processus :

- Possède un processus père
- Peut lancer plusieurs processus fils
- Hérite de presque toutes les caractéristiques de son père : entrée / sortie standard, utilisateur, priorité, ...
- Peut s'exécuter en parallèle avec ses processus fils => le système gère un **arbre de processus**

Les processus : exemple d'arbre

```
$ pstree -pu
init(1)-+-alarmd(888,doe)
        |-apache(763)-+-apache(768,www-data)
        |              `-apache(769,www-data)
        |-atd(756,daemon)
        |-bdflush(5)
        |-cron(759)
        |-cupsd(625)
        |-gdm(778)---gdm(782)-+-XFree86(783)
        |                     `-kde2(815,doe)-+-ksmserver(850)
        |                                     `-ssh-agent(819)
        |-getty(788)
        |-gpm(629)
        |-inetd(657)---famd(840,doe)
        |-kdeinit(828,doe)-+-kdeinit(851)
        |                  |-kdeinit(866)-+-bash(875)
        |                  |              `-bash(878)---pstree(917)
        |                  `-kdeinit(869)---bash(871)
        |-kdeinit(832,doe)
        |-kdeinit(862,doe)
        …
        `-xfs(729)
```

Les processus Unix

- Identifié par leur **PID** (**P**rocess **ID**entifier).
- Le contexte d'exécution contient (entre autres) :
  - Le propriétaire du processus (pour le contrôle des droits)
  - La priorité du processus (pour l'ordonnancement)

États principaux (un seul à la fois) :

- **Élu** : en cours d'**exécution**.
- **Prêt** : en attente du processeur.
- **Bloqué** : en attente de résultats (Entrées/Sortie, I/O).

L'ordonnancement des processus

L'ordonnanceur

- Sélectionne le(s) processus disposant du (ou des) processeur(s).
  - Donc change les états des processus.

POSIX définit les politiques d'ordonnancement suivantes :

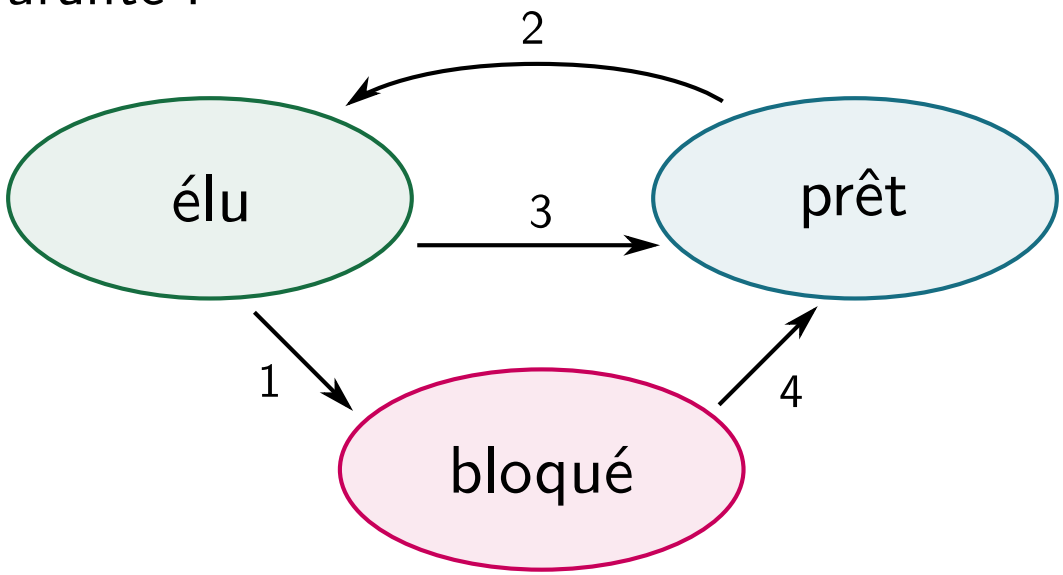
- *Temps partagé*, sans contraintes temporelles.
- *Temps réel*, avec des contraintes temporelles.

L'ordonnanceur «classique» d'Unix (1/2)

- Mode *temps partagé*.
- Donne l'illusion à l'utilisateur qu'il dispose seul du processeur.
  - Choisit le processus **prêt** dont la *priorité* est la plus grande.
  - Priorité basée sur la valeur de **gentillesse** (*niceness*) du processus (cf. commande `nice`).
    - → *Niceness* augmente chaque fois que le processus n'est pas sélectionné alors qu'il est prêt.
  - Ordonnanceur **préemptif** (par opposition aux systèmes *coopératifs*).

L'ordonnanceur «classique» d'Unix (2/2)

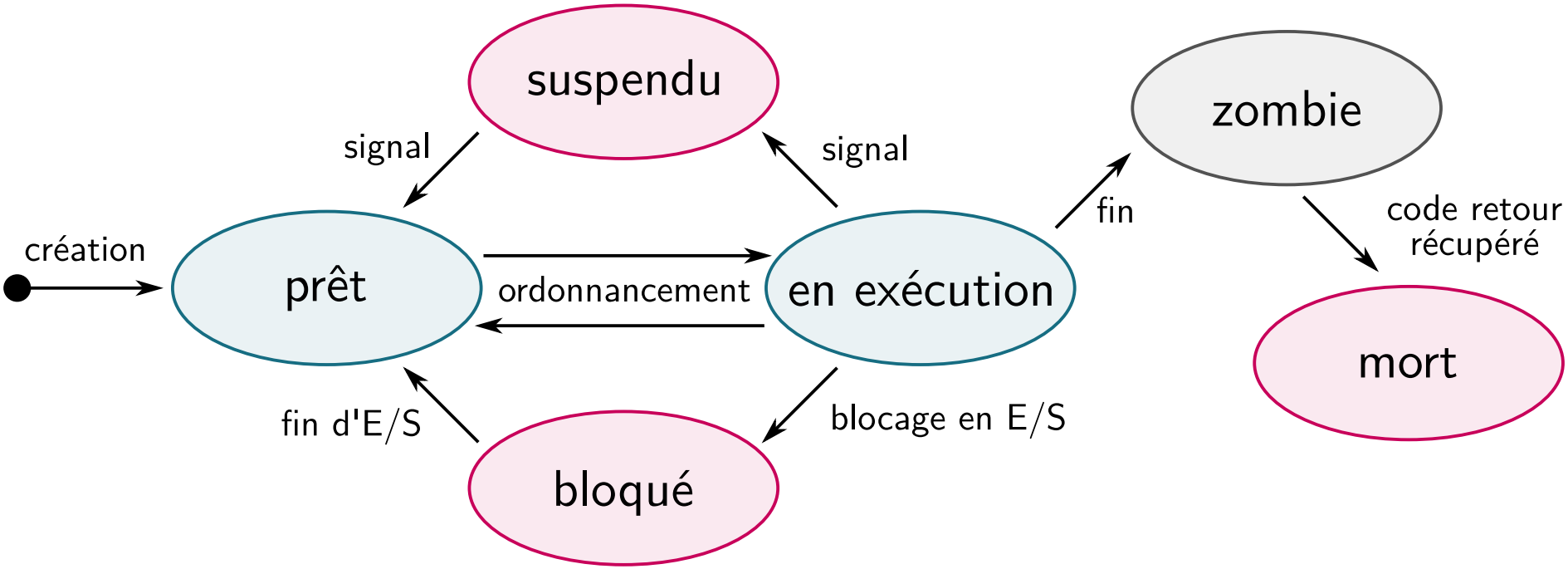
Une situation courante :



- Les transitions sont les suivantes :
  1. Le processus effectue une opération bloquante (E/S par ex.) et attend les résultats
  2. Le processus le plus prioritaire récupère le processeur
  3. Le processus a atteint son temps d'utilisation CPU et est remis dans la liste des processus à ordonnancer
  4. Le processus a reçu les résultats de l'opération bloquante

Les états (détaillés) d'un processus

Ce que vos parents vous cachent...



La commande `ps`

- `ps` permet d'afficher l'état de processus en cours

```
$ ps auwx
USER     PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
…
root     756  0.0  0.1   1320   416 ?        S    Sep03   8:44 /usr/sbin/lpd
…

$ ps -efl
  F S UID       PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY      TIME CMD
…
100 S root    18547 18546  5  68 -10 - 28724 select Sep12 ?   06:41:07 /usr/…
…
```

Le code de retour des processus (1/2)

- Un processus qui se termine génère une *valeur de retour* (code de retour).
  - **0** => terminé normalement
  - **!= 0** => erreur (valeur -> indication sur l'erreur)
- Stocké par le shell lanceur dans la *variable* `$?`
  - Change à chaque nouveau processus terminé.
- `exit XXX` renvoie le code *XXX* (avec 0 <= `XXX` <= 255)

Le code de retour des processus (2/2)

Exemples :

```
1 > ls fic # le fichier fic existe
2 fic
3 > echo $?
4 0
5 > ls badfic # le fichier badfic n'existe pas
6 >/bin/ls: badfic: No such file or directory
7 > echo $?
8 1
```

Remarque : Détail des erreurs indiquées dans le code de retour dans le `man`uel

L'enchaînement des commandes shell (1/3)

- `commande`
  - Commande exécutée.
  - Le shell attend la terminaison de la commande pour continuer
- `commande &`
  - Commande exécutée en *arrière-plan*.
  - Le shell n'attend pas la fin de la commande pour continuer.
- `commande_1; commande_2; … ; commande_N`
  - Lancement séquentiel des *N* commandes
  - La commande *i+1* n'est exécutée que lorsque commande *i* est terminée.
- `commande_1 | commande_2 | … | commande_N`
  - exécution *concurrente* (en parallèle) des N commandes.

L'enchaînement des commandes shell (2/3)

- `(commande)`
  - Commande exécutée dans un *sous processus shell*.
- `commande_1 && commande_2`
  - `commande_2` est exécutée **si et seulement si** le code de retour de `commande_1` **est nul**.
- `commande_1 || commande_2`
  - `commande_2` est exécutée **si et seulement si** le code de retour de `commande_1` **n'est pas nul**.

Remarque : le code de retour d'une commande composée est le code de retour du dernier processus exécuté.

L'enchaînement des commandes shell (3/3)

Exemples :

- `date | grep "Sun" > /dev/null || echo 'Au boulot !'`
  - affiche "Au boulot" si le jour de la semaine n'est pas dimanche (grep à un code de retour égal à 1 (faux) si aucune correspondance n'est trouvé)
- `(date; quota -v ) > fic`
  - la sortie standard du processus shell créé est redirigée vers le fichier fic. Toutes les commandes exécutées par ce processus auront leur sortie standard redirigée vers fic (héritage des E/S)
- `(echo "test"; exit 5)`

*un sous processus shell est créé. Il affiche "test" et se termine en retournant 5

Les variables en shell (1/4)

- Nom de variable
  - Commence par une lettre
  - Composé de caractères alphanumériques.
  - Longueur maximale varie suivant les shells.
- Type par défaut : chaîne de caractères.

Deux catégories :

1. *Variables locales au shell*.
   - `set` : affiche les variables locales existantes.
2. *Variables d'environnement* (*exportées*) : transmises par valeur aux processus fils.
   - `env` affiche les variables d'environnement existantes.

Les variables en shell (2/3)

Affectation : `nom=valeur`

Attention

Pas d'espace autour du `=`

- `export nom` : transforme une variable en *variable d'environnement*.
  - `export nom=valeur` : Affecte et exporte en même temps.

Remarque : Les variables sont *passées par valeur* (par copie) => un processus fils ne peut jamais modifier le contenu d'une variable de son père.

Les variables en shell (3/3)

- Accès au contenu de la variable `nom` : `$nom`

Exemple :

```
1 a=1
2 echo $a  # Affiche 1
3 b=$a+1
4 echo $b
5 b=$(($a+1)) # pour forcer l'évaluation arithmétique
6 echo $b
```

- `unset nom` “dé-affecte” une variable.

Variables shell relatives aux processus

- `?` : le *code de retour* du dernier processus terminé.
- `$` : le *PID du shell courant*.
  - Dans le cas d'un clone (sous processus créé via `()`) : retourne le PID du shell du père.
- `PPID` : le *PID du processus père*.
  - Dans le cas d'un clone (sous processus créer via `()`) retourne le PID du processus grand-père.
- `!` : le PID du *dernier processus lancé en arrière-plan*.

Les signaux

Entier adressé à un processus pour l'avertir d'un événement. L'émetteur parmi :

- Noyau
- Autre processus
- Utilisateur par l'intermédiaire du clavier.

```
$ /usr/bin/kill -L
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 2) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 3) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
1)  SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
2)  SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
3)  SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
4)  SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
```

Réception d'un signal

Actions exécutées à la réception :

1. terminaison du processus
2. terminaison du processus avec image mémoire (fichier core)
3. signal ignoré (sans effet)
4. suspension du processus (il s'endort)
5. continuation : reprise d'un processus stoppé et ignoré sinon (se réveiller)

`trap [-l] [arg] [sigspec]` place un nouveau «handler» pour associer un comportement à la réception d'un signal donné (sauf pour les signaux `SIGKILL`, `SIGCONT` et `SIGSTOP`).

Signaux POSIX

- Le chiffre entre crochets indique le comportement par défaut.
- - indique un comportement non modifiable.
  - `SIGHUP` [1] : terminaison du processus leader de session
  - `SIGINT` [1] : frappe du caractère `intr` (`CTRL + c`) sur le clavier du terminal de contôle
  - `SIGQUIT` [2] : frappe du caractère `quit` (`CTRL + \`) sur le clavier du terminal de contrôle
  - `SIGILL` [2] : détection d'une instruction illégale
  - `SIGABRT` [2] : terminaison anormale provoquée par l'exécution de la fonction abort

Encore des signaux

- `SIGFPE` [2] : erreur arithmétique (division par zéro, ...)
- `SIGKILL` [1*] : signal de terminaison
- `SIGSEGV` [2] : violation mémoire
- `SIGPIPE` [1] : écriture dans un tube sans lecteur
- `SIGALRM` [1] : fin de temporisation (fonction alarm)
- `SIGTERM` [1] : signal de terminaison
- `SIGUSR1` [1] : signal libre (sans signification prédéfinie)
- `SIGUSR2` [1] : signal libre (sans signification prédéfinie)
- `SIGCHLD` [3] : terminaison d'un fils

Les signaux de contrôle des processus

- `SIGSTOP` [4*] : signal de suspension
- `SIGTSTP` [4] : frappe du caractère `susp` (`CTRL + z`) sur le clavier du terminal de contrôle
- `SIGCONT` [5*] : signal de continuation d'un processus stoppé
- `SIGTTIN` [4] : lecture par un processus en arrière-plan
- `SIGTTOU` [4] : écriture par un processus en arrière-plan (terminal en mode tstp)

Autres signaux

- `SIGIO` [1] : avis d'arrivée de caractères à lire
- `SIGURG` [3] : avis d'arrivée de caractères urgents (POSIX.1-2001)
- `SIGWINCH` [3] : redimensionnement de fenêtre

Émission d'un signal

- `kill [-sigspec] pid…` émet `sigspec` vers `pid`.
- `killall [-sigspec] nom…` émet `sigspec` vers les processus en train d'exécuter les commandes mentionnés.
  - `killall` n'est pas une commande standard.
- Si `sigspec` n'est pas précisé, `TERM` est envoyé.

```
1 # Envoi du signal SIGINT au processus de pid 1234
2 kill -SIGINT 1234
3 # Envoi de SIGKILL à tous les processus firefox-esr
4 killall -9 firefox-esr
```

Ce programme est-il tuable ?

```
1 #!/bin/bash
2 trap 'echo "Je suis immortel !"' SIGINT SIGQUIT
3 i=0
4 while [ 0 ]; do
5     echo $i
6     i=$(($i+1))
7     sleep 1
8 done
```

Timeout en shell

```
1 #!/bin/bash

2 trap 'echo "Le temps est écoulé !"; exit 1' SIGALRM

3 (sleep 3; kill -SIGALRM $$ )&
4 echo -n "Vous disposez de 3 s. pour entrez un mot: "
5 read MOT
6 trap - SIGALRM; kill -SIGKILL $! 2> /dev/null
7 echo "le mot lu: $MOT"
```

Expressions rationnelles

Les expressions rationnelles (*regular expressions*)

Les expressions rationnelles (`regex`) sont des motifs (*pattern* en anglais) permettant de décrire des chaînes de caractères.

- Décrire une chaîne de caractères pour la rechercher, la remplacer, la supprimer, etc. avec les commandes `grep` , `sed` , ...
- Utilise des caractères génériques, des ensembles, des répétitions, pour analyser un texte et détecter des motifs, vérifier des syntaxes, ...
- Utilisable dans certaines commandes mais aussi dans la plupart des langages de programmation

Décrire un caractère – un atome

- Un caractère spécifique : le caractère lui même `a`, `û`, `0`, ...
- N'importe quel caractère : utilisation du joker `.`
- Un caractère choisi dans une liste : `[«list»]`
  ex : `[012345678]` tous les *chiffres* sauf `9`
- Un caractère choisi hors d'une liste : `[^«liste»]`
  ex : `[^9]` tous les *caractères* sauf `9`
- Un caractère de l'intervalle : `[«debut»-«fin»]`
  `[a-z]` : une minuscule
  `[^A-Z]` : qui n'est pas une majuscule

Attention

Attention à la valeur de la locale `LC_COLLATE` (`export LC_COLLATE=C`)

Décrire un caractère – un atome

Possibilité d'utiliser des ensembles prédéfinis :

- `[:alnum:]` : alpha numériques `a-zA-Z0-9`
- `[:digit:]` : chiffres ou `0-9`
- `[:xdigit:]` : les chiffres hexadécimaux `0-9a-fA-F`
- `[:lower:]` : `a-z`
- `[:upper:]` : `A-Z`
- `[:alpha:]` : `A-Za-z`
- `[:space:]` : les caractères d'espacements (espace et tab)

Attention

N'oubliez pas de crochets ! Par exemple, l'équivalent de la lettre minuscule `[a-z]` est `[[:lower:]]` puisque `[:lower:]` est équivalent à `a-z`.

Les répétitions d'atomes

Les répétitions suivent les atomes pour indiquer combien de fois ce dernier atome se répète dans le motif :

- `?` : l'atome est optionnel (présent 0 ou 1 fois)
- `*` : l'atome est optionnel et peut se répéter (0 ou n fois)
- `+` : l'atome est obligatoire et peut se répéter (1 fois ou plus)
- `{n}` : l'atome est présent exactement n fois
- `{n,}` : l'atome est présent au moins n fois
- `{,m}` : l'atome est présent au plus m fois (remarque : ne fonctionne pas avec tous les outils)
- `{n,m}` : l'atome est présent entre n et m fois

Exemple : un mot est une suite d'au moins un caractère alphabétique séparée par des espaces : `␣[A-Za-z]+␣` ou mieux

```
[:space:][[:alpha:]]+[:space:]
```

Quelques caractères spéciaux

- Indiquer le début d'une ligne : `^`
- Indiquer la fin d'une ligne : `$`
- Indiquer le début ou la fin d'un mot : `\b` (`\b` correspond à une chaîne vide à l'extrémité d'un mot)
- Protection des caractères : préfixer par `\`
  Pour indiquer par exemple le caractère `*` il faudra écrire `\*`

Un premier exemple

Décrire une ancienne plaque d'immatriculation française (métropole) : 1234 AB 63

- 1 chiffre entre 1 et 9 suivi d'au plus 3 chiffres entre 0 et 9 (nombres de 1 à 9999)
- 1 à 3 lettres majuscules
- 2 chiffres pour le département

```
[1-9][0-9]{,3}[A-Z]{1,3}[0-9]{2}
```

Petit hic : le département 00 n'existe pas...

Un deuxième exemple

Décrire un nombre relatif : par exemple `+2.1` `-15` `6.7` `+3` ...

- Un signe + ou - suivi d'au moins un chiffre suivi éventuellement d'un point et de chiffres `[+-]?[0-9]+\.?[0-9]+`
  - `[+-]?` => un signe facultatif, présent au plus une fois
  - `[0-9]+` => au moins un chiffre
  - `\.?` => le symbole `.` présent au plus une fois
  - `[0-9]+` => au moins un chiffre dans la partie décimale

Petit hic : `2` ne correspond pas. Il nous manque quelque chose...

Les expression alternatives

- Décrire plusieurs possibilités :
  `( possibilité_1 | possibilité_2 |  … )`
- `|` indique un « *ou* »
- Les nombres relatifs deviennent : `[+-]?([0-9]+|[0-9]+\.[0-9]+)`

Remarque : .56 et ab2c sont reconnus car toutes lignes contenant au moins un chiffre correspondent à `[0-9]+`. Il faut donc délimiter notre motif en précisant par exemple qu'il est en début de ligne `^[+-]?([0-9]+|[0-9]+\.[0-9]+)` ou en début d'un mot `\b[+-]?([0-9]+|[0-9]+\.[0-9]+)`.

Les groupes

- Exemple de la plaque :
  `[1-9][0-9]{0,3}[A-Z]{1,3}([1-9][0-9]|0[1-9])`
- Les groupes définis par `(«groupe»)` peuvent être utilisés pour indiquer des répétitions
- Décrire que l'on recherche une suite de deux groupes constitués de 3 chiffres suivis de 2 caractères : `([0-9]{3}[A-Z]{2}){2}`

Plus loin avec les groupes

- Il est possible de décrire la répétition d'un groupe en utilisant `\n` où *n* est le numéro du groupe
- Par exemple pour indiquer une répétition d'un même caractère dans une phrase :
  `(.)\1` => n'importe quel caractère suivi du même
- Trois caractères qui se suivent dans un sens puis dans le sens inverse plus loin dans la ligne (est...tse) :
  `(.)(.)(.).*\3\2\1` => 3 caractères suivis de n'importe quoi puis suivi du même groupe inversé de 3 caractères

Les outils

En TP nous utiliserons les commandes suivantes :

- `grep` qui affiche les lignes correspondant à un motif donné. Nous l'utiliserons avec son option `-E`
- `sed` qui est un éditeur de flux permettant le filtrage et la transformation de texte. Nous l'utiliserons essentiellement dans sa syntaxe pour faire des substitutions de texte.

`sed -E 's/([0-9]+)/[\1]/g'` : cherche dans stdin toutes les suites de chiffres et les entoure de crochets.
`sed -Ee 's/([0-9]+)/[\1]/g -Ee 's/ +/ /g'` : remplace encadre les chiffres *et* supprime les espaces multiples.

- `awk` qui est un puissant langage de manipulation de texte

Quiz

- Quelles lignes sont reconnues par la regex suivante ?

```
^a(ab)*a$
```

1. abababa
2. aaba
3. aabbaa
4. aba
5. aabababa

Quiz

- Quelles lignes sont reconnues par la regex suivante ?

```
^ab+c?$
```

1. abc
2. ac
3. abbb
4. abbbcc
5. bbc

Quiz

- Quelles lignes sont reconnues par la regex suivante ?

```
^a.[bc]+$
```

1. abc
2. abbbbbbbb
3. azc
4. abcbcbcbc
5. ac
6. asccbbbbcbcccc

Quiz

- Quelles lignes sont reconnues par la regex suivante ?

```
^abc|xyz$
```

1. abc
2. abcyz
3. abc|xyz
4. xyz
5. abxyz

Quiz

- Quelles lignes sont reconnues par la regex suivante ?

```
^[a-z]+[.?!]$
```

1. battle !
2. Hot
3. green
4. swamping.
5. jump up.
6. undulate ?
7. is. ?

Quiz

- Quelles lignes sont reconnues par la regex suivante ?

```
^[a-zA-Z]*[^,]=$
```

1. Butt=
2. BotHEr,=
3. Ample
4. FIdDlE7h=
5. Brittle␣=
6. Other.=

Quiz

- Quelles lignes sont reconnues par la regex suivante ?

```
^[a-z][.?!]\s+[A-Z]$
```

1. `A. B`
2. `c! d`
3. `e f`
4. ‘g. H‘
5. `i?  J`
6. `k L`

Quiz

- Quelles lignes sont reconnues par la regex suivante ?

```
^(very )+(smart )?(tall|ugly) man$
```

1. very smart man
2. smart tall man
3. very very smart tall man
4. very very smart tall ugly man
5. very very very tall man

Quiz

- Quelles lignes sont reconnues par la regex suivante ?

```
^<[^>]+>$
```

1. `<an xml tag>`
2. `<opentag> <closetag>`
3. `</closetag>`
4. `<>`
5. `<with attribute="77">`

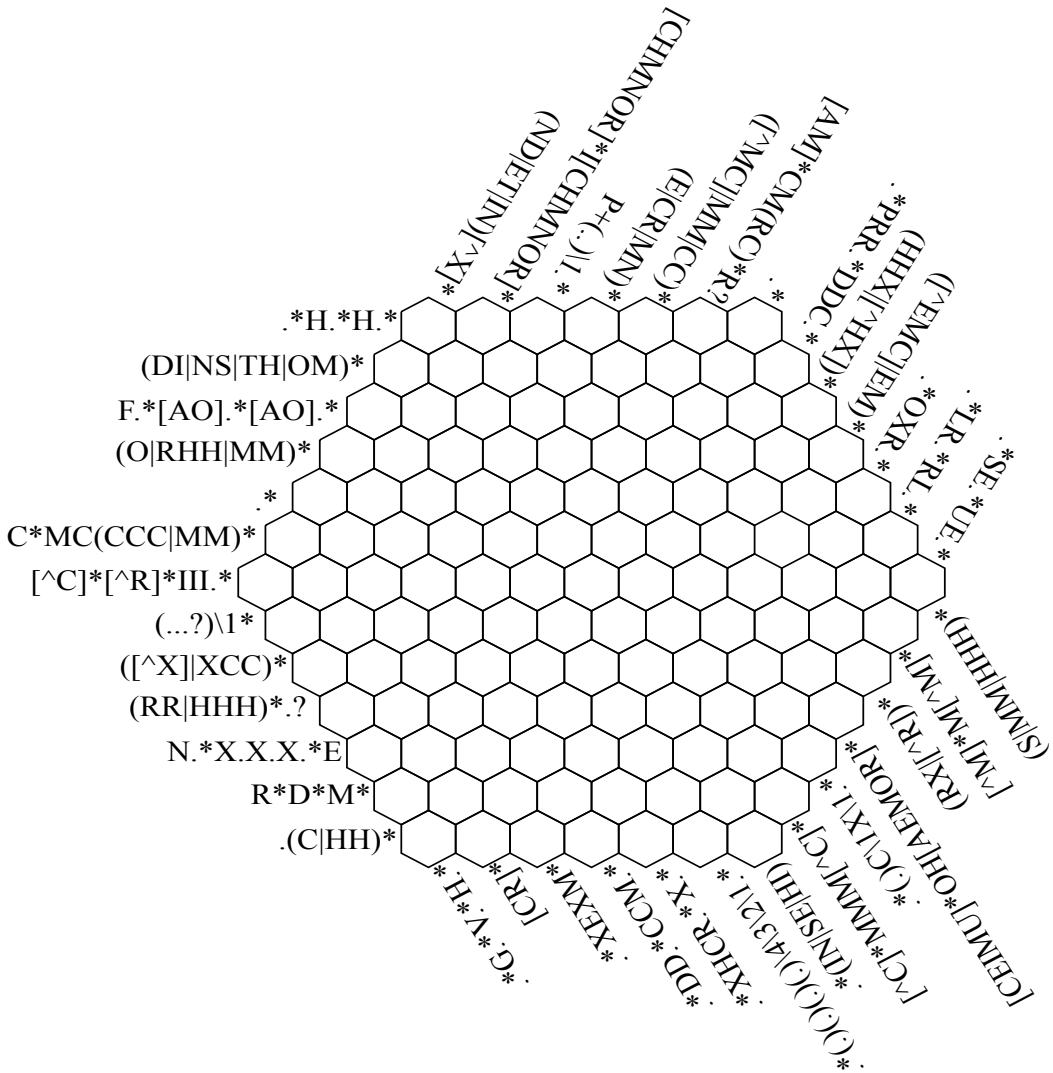
Regex croisées

	`(N|A)*`	`(FO|A|R)*`	`(D|FU|UF)+`	`[^NRU](NO|ON)`
`[RUNT]*`				
`O.*[HAT]`				
`(.)*DO\1`				

Regex croisées (un autre)

	`[ARK]*O.*`	`(.).*\1N\1`	`(SOD|DO|GE)*`	`[FAXUS]*`	`[LOPITY]*`
`[UGLER]*`					
`[CAST]*REX[PEA]*`					
`[SIRES]*`					
`(L|OFT|ON)*`					
`H*(AY|ED)*`					

Regex croisées (un peu plus complexe)



Construisez une expression rationnelle :

1. Permettant de valider le format d'une date
   `JJ/MM/AAAA` ou bien `JJ-MM-AAAA`
   - rappel : 31 jours max et 12 mois dans une année
   - on ne validera pas la cohérence de la date, ex : 31/02/0001

2. Permettant de valider une adresse IPv4
   `192.168.109.5, 127.0.0.1, …`
   - les 4 nombres appartiennent à l'intervalle [0-255]

Construisez une expression rationnelle :

3. Permettant de reconnaître une adresse email

`linus.torvalds@uca.fr, john+doe@unknown.org, …`

   - les constituants ne pourront pas commencer par un chiffre
   - les constituants ne pourront pas être que des numéros
   - le nom de domaine devra contenir au moins un `.` et ses 2 parties seront non vides

4. Permettant de valider une commande de connexion SSH

`ssh [user@]host[:/path/to/file]`

   - on suppose la commande sans options
   - on suppose que `user` n'a que des caractères alpha-numériques

Les quotes

- Un texte entre simples quotes `'` est interprété *littéralement*
  - Aucune substitution !
  - `a=42; echo '$a'` affiche `$a`
- Un texte entre doubles quotes `"` est interprété littéralement sauf les caractères `` ` `` (accent grave, antiquote), `$` et `"`
  - Pratique pour protéger du texte tout en évaluant les variables
  - `""` : chaîne vide (ex : `"$varExistePas"`)
- Antiquote `` ` `` (ou backquote)
  - `` res=`ls -l $fichier` `` équiv `res=$(ls -l $fichier)`
  - Évalue son contenu *après* expansion des variables
  - Capture la sortie
  - Préférer `$()`
- Backslash (`\`)
  - Protège les caractères spéciaux

Exemples sur les quotes

- Que valent les variables suivantes :
  - `a=$(date)`
  - `b=$(ls ~/ | grep '\.[ch]$' | wc -l)`
  - `c='$PATH : liste des répertoires des exécutables'`
  - `d="nous sommes le $a"`
  - `e="Nous sommes le $(date)"`
  - `f=$($d)`

- `cc='gcc -Wall -ansi'`
  - `if $($cc "mon prog.c" -o exec); then echo ok; fi`

Les variables (rappels)

- Variables locales
  - Non transmises au processus fils lancés
  - Fixées par l'affectation (`=`)
  - Listées par la commande `set` sans argument

- Variables exportées (dites « d'environnement »)
  - Transmises aux processus fils
  - Fixées par `set -a` , ou exportées par `export nomVar`
  - Listées par la commande `env`
  - `unset varName` : suppression de déf d'une variable

Les scripts Shell

- Un fichier contenant une succession de commandes shell
  - Peut s'exécuter (être interprété)
- Première ligne (**shebang**) : nom du shell à exécuter
  - `#!/bin/bash`
  - `#!/usr/bin/env bash` (plus portable)
- Doit être exécutable si on veut l'appeler directement
- Appel comme n'importe quelle commande
  - `monProg arg1 arg2`
  - Beaucoup de programmes d'un système Unix sont des scripts
  - Attention au chemin d'exécution (`PATH`)
- `# Ceci est un commentaire`

Exemples simplissimes

- Hello world

```
1 #!/bin/sh
2 echo 'Hello World'
```

- Copier les TP de C sur dans /tmp

```
1 #!/bin/sh
2 find ~ -iname '*.[ch]' -exec cp {} /tmp \;
```

Les arguments des scripts

- `$1` : Premier argument,
- `$2` : Second argument (et ainsi de suite)
- `$*` ou `$@` : Tous les arguments
- `$#` : Nombre d'arguments
- `$0` : Nom du shell ou du script
- `shift` : Supprime le 1er arg. et décale les autres
- Rappels :
  - `$$` : PID du shell
  - `$!` : PID du dernier fils
  - `$?` : Valeur de retour de la dernière commande

Utilisation des arguments, exemple

- Programme

```
1 #!/bin/bash
2 # j'affiche mon deuxième argument, suivi de mon premier
3 printf "$2, $1"
```

- Appel

```
1 ./affArg 'Je suis le premier argument' \
2          Et il y en a d\'autres
```

- Affichage ?

if… then… fi

```
1  if commande_1
2    then
3      instruction_1
4      …
5      instruction_N
6    elif commande_2; then
7      …
8    else
9      …
0  fi
```

Important

- Si `if` et `then` sont sur la même ligne, les séparer par un `;`
- Espaces **avant** et **après** les `[]`

if... then... fi (exemple)

```
1 if [ -f "$1" ]
2   then
3    echo "C'est un fichier"
4    ls -l "$1"
5   else
6     echo "Mais qu'est-ce ?"
7 fi
```

Les tests

- Cas général : `if` se base sur le code de retour de la commande
- Cas particulier : les commandes `[` et `test` permettent de réaliser des tests

`test expression` ou alors `[ expression ]`

- Retourne 0 si *vrai*, une autre valeur dans les autres cas
- Exemples :
  - `if test 3 -eq 3 ; then …`
  - `if [ 3 -eq 3 ] ; then …`
  - `if test $1 = $2 ; then …`
  - `if [ $1 = $2 ] ; then …`

Tests sur chaînes et entiers

- Test sur les chaînes
  - `s1 = s2`, si les chaînes `s1` et `s2` sont identiques
  - `s1 != s2`, si les chaînes sont différentes.
  - opérateurs
    - → `-z` : chaîne vide
    - → `-n` : chaîne non vide
- Test sur les entiers
  - `n1 -eq n2`, si `n1` est égal à `n2`
  - opérateurs `-ne`, `-eq`, `-gt`, `-lt`, `-le`, `-ge`

Tests sur fichiers

- Les plus utiles :
  - `-d f`, si `f` est un répertoire
  - `-e f`, si le fichier `f` existe
  - `-f f`, si `f` est un fichier simple
  - `-x f`, fichier `f` existe ET exécutable
  - `-w f`, fichier `f` existe ET écriture autorisée
  - `-r f`, fichier `f` existe ET lisible
- Et plus encore (voir le manuel)

Tests variés

- Inversion de test

```
1 i=0;
2 while [ ! $i -eq 10 ]; do echo $i; i=$(($i+1)); done
```

- Utilisation de commande (code de retour)

```
1 if cp "$fichier" /tmp; then echo 'fichier copié'; fi
```

- Chaînage de commandes sur code de retour
  - TOUJOURS la valeur de la dernière commande
  - `cmde_1; cmde_2` : `cmde_2` toujours évaluée
  - `cmde_1 && cmde_2` : `cmde_2` effectuée ssi `cmde_1` OK
  - `cmde_1 || cmde_2` : `cmde_2` effectuée ssi `cmde_1` KO

Exemples de tests

- Si le fichier peut être lu et écrit :

```
1 if [ -r fic ] && [ -w fic ] ; then …; fi
```

- Si `chaine` présente dans `/etc/passwd` ou `/etc/group` :

```
1 if grep -q 'chaine' /etc/passwd
2 || grep -q 'chaine' /etc/group
3 then
4   …
5 fi
```

Exemples de tests

- Si le fichier n'existe pas :

```
1 if [ ! -e "$fichier" ] ; then echo "$fichier manquant"; fi
```

- Si le nombre est > 10 ou < 3 :

```
1 if [ $nb -gt 10 ] || [ $nb -lt 3 ] ; then …; fi
```

Exos

1. Écrire un script qui reçoit en paramètre l'heure :

```
./monscript 09 30 45 # pour 9h30 et 45 secondes
```

   - Et qui retourne
     - 0 si l'heure est au bon format (heure $\in$ [0,23], minutes $\in$ [0,59] et secondes $\in$ [0,59])
     - Retourne 1 sinon et affiche sur la sortie erreur : "format incorrect"

2. Écrire un script qui retourne 0 si le nombre passé en paramètre est pair, 1 sinon

Solution heure

```
1 #!/bin/bash
2 ret=0
3 if [ $1 -lt 0 ] || [ $1 -gt 23 ] ; then ret=1 ; fi
4 if [ $2 -lt 0 ] || [ $2 -gt 59 ] ; then ret=1 ; fi
5 if [ $3 -lt 0 ] || [ $3 -gt 59 ] ; then ret=1 ; fi
6 if [ $ret -eq 1 ] ; then
7     echo "Format incorrect" >&2
8 fi
9 exit $ret;
```

Solution parité

```
1 #!/bin/bash
2 nb=$(( $1 / 2 ))
3 nb=$(( $nb * 2 ))
4 if [ $nb -eq $1 ] ; then
5     exit 0
6 else
7     exit 1
8 fi
```

Les boucles

- Répéter jusqu'à

```
1 until condition
2 do
3   commandes
4 done
```

- Tant que

```
1 while condition
2 do
3   commandes
4 done
```

Les boucles (2/2)

- Pour tous les éléments d'une liste

```
1 for i [ in mots ]
2 do
3   commandes
4 done
```

- Si `in mots` n'est pas précisé, `$1`, `$2`, ... sont utilisés.

Exemples de boucles

```
i=0; until [ $i -eq 10 ] ; do
  i=$(($i+1))
done
```

```
for i in `seq 1 10`; do
  echo $i
done
```

```
for fruit in pomme poire banane; do
  echo $fruit;
done
```

```
for fichier in $(ls ~); do # Préférer *
  echo "Fichier $fichier: `file $fichier`"
done
```

Exemples (suite)

- Se connecter sur troll dès que celui-ci est booté :

```
1 until ssh troll; do sleep 30; done
```

- L'horloge la plus simple :

```
1 clear; while date; do sleep 1; clear; done
```

- Afficher un message dès que `/tmp/coucou` est créé :

```
1 fichier='/tmp/coucou'
2 until [ -e $fichier ]; do sleep 2; done
3 echo "$fichier a été créé"
```

Exos boucles

1. Écrire un script effectuant la copie de tous les fichiers d'un répertoire passé en paramètre dans le répertoire courant.
2. Écrire un script comptant le temps : il « compte » les secondes puis les minutes puis les heures dans 3 boucles imbriquées et affiche chaque seconde la nouvelle heure. Avec 3 types de boucles.

Solution copie

```
1 #!/bin/bash
2 for fic in `ls $1` ; do
3     if [ ! -d "$1/$fic" ] ;then
4         cp "$1/$fic" .
5     fi
6 done
```

Solution fausse horloge

```
1  #!/bin/bash
2  while [ 1 -eq 1 ] ; do
3      for h in `seq 1 24` ; do
4          m = 0
5          while [ $m - lt 60 ] ; do
6              s=0
7              until [ $s -eq 60 ] ; do
8                  echo $h:$m:$s
9                  s=$(( $s + 1))
0                  sleep 1
1              done
2              m=$(( $m + 1 ))
3          done
4      done
5  done
```

Diverses commandes (1/2)

- Lecture d'une chaîne sur entrée standard : `read`

```
1 echo -n "Entrez votre nom :"
2 read nom
3 echo "Votre nom est $nom"
```

- `read` sans variable -> résultat dans `REPLY`
- `read a1 a2 a3` -> premier mot dans a1, deuxième mot dans a2 et **tous les autres** dans a3

Diverses commandes (2/2)

- Evaluation mathématique
  - `i=$(expr $i + $1)` (commande externe... lent)

```
time ./essais.sh # 10.000 iteration
49,99s user 13,02s system 99% cpu 1:03,29 total
```

- `i=$(($i + 1))` (peu portable mais plus rapide)

```
time ./essais.sh # 10.000 iteration
0,32s user 0,01s system 103% cpu 0,319 total
```

- Commande externe bc pour calcul en flottants :
  - `r=$(echo "scale=8000; sqrt(2)" | bc)`

Exercices

1. Écrivez un script `num` acceptant en argument une liste d'entiers et qui n'affiche que ceux qui sont positifs ou nuls
2. Écrivez le script `alpha` lisant son entrée standard et :
   a. n'affichant que les lignes paires
   b. qui en plus contiennent un nombre d'au moins deux chiffres
   c. en les passant en majuscules (via la commande `tr a-z A-Z`)
   d. en les faisant précéder de leur numéro (1 pour la première ligne)

Les scripts sous Unix (suite)

- `for`
- `break`
- `continue`
- `case`
- fonctions
- `alias`

L'instruction `for`

- Sémantique : *for each*
  - `for i in list ; do commandes; done`
  - `for i in ours elephant castor; do echo $i; done`
  - `for i in *; do echo $i; done`
  - `var='a b c d e'; for i in $var; do echo $i; done`
  - `for i in `ls /bin`; do echo $i; done`
  - `for i in `seq 1 10`; do echo $i; done`
- `IFS` : liste des séparateurs
  - `IFS=':';var='ab:b c:d'`
    `for i in $var; do echo $i; done`
  - `i` prend donc pour valeurs : `ab` puis `b c` puis `d`

`break` : interrompre une boucle

- `break` : quitte la boucle (`for` , `while` , `until` , `select`)

```
1 for i in a b c; do
2     if [ $i = 'b' ]; then
3         break
4     fi
5     echo $i
6 done
```

Exemple de `break`

```
while [ $# -gt 0 ] ; do
    if [ "$1" = '-v' ] ; then
        echo "verbose"
    elif [ "$1" = '-m' ] ; then
        echo "mode m"
    else
        echo "erreur de syntaxe, option inconnue !" >&2
        break;
    fi
    shift
done
```

`continue` : continuer les itérations d'une boucle

- `continue` : continue la boucle (`for` , `while` , `until`) en sautant le traitement suivant

```
1 for i in a b c; do
2     if [ $i = 'b' ]; then
3         continue
4     fi
5     echo $i
6 done
```

Exemple de `continue`

```
1 for i ; do
2     reste=$(( $i % 2 ))
3     if [ $reste -ne 0 ] ; then
4         continue
5     fi
6     echo $i
7 done
```

Instruction `case`

- `case word in`
  `[ [(] pattern [ | pattern ] … ) list ;; ] …`
  `esac` : selon certaines valeurs

```
1  while [ $# -ne 0 ]; do
2     case "$1" in
3        '-v')
4           echo 'Mode verbose'
5           shift
6           ;;
7        '-m'|'-M')
8           machine="$2"
9        shift 2 || sortie "Manque valeur"
0        ;;
1         *) sortie "Argument $1 inconnu"
2        ;;
3     esac
4  done
```

Les fonctions (1/2)

- Presque comme un script shell indépendant

```
1  #! /bin/bash
2  #Déclaration de la fonction "maFonction"
3  maFonction() {
4      #Accès aux arguments de la fonction
5      echo "Mes arguments sont $*"
6      return 3
7  }
8  # Programme principal
9  maFonction lapin ours toto
10 echo Le code de retour de la fonction est $?
```

Fonctions (2/2)

- Accès aux arguments par `$1`, `$2`, ..., `$*`
- Connaît toutes les variables de la fonction appelante !
  - Mais, mieux vaut ne pas les utiliser (portabilité !)
- Retour de la fonction par `return value` (entier)
- Appel comme une commande (sans `()`)
- Hérite des entrées/sorties standard du père
- Ne crée pas de processus sauf si appelée avec `|`
  - Exemple : `cat /etc/passwd | maFonction`

Variables & fonctions

- Par défaut, une variable shell est **globale**
  - Donc visible pendant TOUTE l'exécution du script !
- Variables globales = cauchemar du programmeur
- Donc déclaration de variables **locales**
  - Définies dans les fonctions
  - Masque les variables globales
- Déclaration par `local nomVariable`
- Variables locales « à la mode shell », visibles dans toutes les sous-fonctions appelées !
  - Solution : faire attention et ne pas les utiliser

Variables locales : exemple

```
1  #!/bin/bash

2  fonctionFille() {
3      echo Dans fille: $varMere $varLocMere
4      local varLocFille
5      varLocFille="varLocFille"
6      varFille="varFille"
7  }

8  fMere() {
9      local varLocMere="varLocMere"
0      varMere="varMere"
1      fonctionFille
2      echo Dans mere: $varLocFille $varFille
3  }

4  fMere
```

Communiquer avec les fonctions

- Entrée
  - Arguments de la fonction (`$1` , ..., `$*`)
  - Variables des fonctions de plus haut niveau (beurk !)
- Sortie
  - Argument de retour (`return 0` / `echo $?`)
    - → Utilisé souvent pour tester le bon fonctionnement de la fonction
    - → Limité à des valeurs entières entre 0 et 127
  - Variables globales (bof)
  - Ecriture sur la sortie standard
    - → `f() { echo "MonResultat: $1"; return 0; }`
    - → `result=`f truc``

Fonctions : E/S standards

- Par défaut, celles du père
  - `getVal() { echo "Entrez valeur"; read VALEUR; }`
  - `getVal; echo $VALEUR`
- Si appelé avec `|` (pipe)
  - `echo "J'ai une question" | getVal; echo $VALEUR`
  - Crée un processus, donc n'affiche rien (sauf avec ksh/zsh) !!
- Solution avec pipe (par sortie standard) :

```
1 getVal() {
2   echo "Entrez une valeur :"; read VAL; echo $VAL`
3 }
4 VALEUR=$(echo "J'ai une question" | getVal)
```

TD fonctions

- Écrire la fonction `min` qui retourne le minimum des arguments passés
- Écrire la fonction `getNumLignes` qui affiche le nombre de lignes du fichier `f` passé en paramètre ; retourne `0` si tout va bien, `1` si le fichier n'existe pas ou n'est pas lisible
- Écrire la fonction `extractLigne` qui affiche la ligne `L` du fichier `f` les arguments étant passés dans cet ordre ; retourne `0` si tout s'est bien passé, `1` si la ligne n'existe pas dans le fichier
- Écrire la commande `cat` utilisant les fonctions ci-dessus

Solution `min`

```
1  min() {
2      if [ $# -eq 0 ]; then
3        return 1
4      fi
5      local mini="$1"; shift
6      local nb
7      for nb; do
8          if [ $mini -gt $nb ]; then
9              mini=$nb
0          fi
1      done
2      echo $mini
3      return 0
4  }
```

Solution `getNumLignes`

```
1 getNumLignes() {
2 if [ $# -eq 1 ] && [ -f "$1" ] && [ -r "$1" ]; then
3     wc -l "$1" | cut -d ' ' -f 1
4     return 0
5 else
6     return 1
7 fi
8 }
```

Solution extractLigne

```
1  extractLigne() {
2  if [ $# -eq 2 ]; then
3      if local maxl=$(getNumLignes "$2") &&
4          [ $1 -gt 0 ] &&
5          [ $1 -le $maxl ]; then
6          cat "$2" | head -n $1 | tail -n 1
7          return 0
8      else
9          return 1
0      fi
1  fi
2  }
```

Solution cat

```
1  #!/bin/bash
2  # fonction getNumLigne
3  getNumLignes() …
4  # fonction extractLigne
5  extractLigne() …
6  for file; do
7      c=1
8      while ligne=$(extractLigne $c "$file"); do
9          echo "$ligne"
0          c=$(( $c + 1 ))
1      done
2  done
```

Les fonctions internes de bash (1/7)

- `: [arguments]` : pas d'effet (pareil que `true`) ; retourne 0
- `. filename [arguments]`

  `source filename [arguments]` : lit et exécute les commandes de `filename` dans l'environnement courant
- `alias [name[=value] …]` : affiche ou définit un alias
- `bg [jobspec]` : place `jobspec` en arrière plan
- `bind [-m keymap] [-lvd] [-q name]`

  `bind [-m keymap] -f filename`

  `bind [-m keymap] keyseq:function-name` : permet de définir l'action associée à une séquence de touches
- `break [n]` : quitte une boucle `for`, `while` ou `until` (ou plusieurs boucles imbriquées si `n` > 1)

Les fonctions internes de bash (2/7)

- `builtin shell-builtin [arguments]` : exécute la fonction interne du shell
- `cd [dir]` : change de répertoire courant
- `command [-pVv] command [arg …]` : exécute ou renseigne sur une commande
- `continue [n]` : reprend à la prochaine itération d'une boucle. Si n est précisé, il indique la boucle imbriquée de niveau inférieur `n`
- `declare [-frxi] [name[=value]]`
  `typeset [-frxi] [name[=value]]` : déclare et/ou fixe les attributs d'une variable
- `dirs [-l] [+/-n]` : affiche la liste des répertoires de la pile (cf. `pushd` et `popd`)
- `echo [-neE] [arg …]` : affiche les arguments

Les fonctions internes de bash (3/7)

- `enable [-n] [name …]` : active ou désactive les fonctions internes du shell
- `eval [arg …]` : exécute (dans l'environnement courant) la commande constituée de la concaténation des arguments
- `exec [[-] command [arguments]]` : remplace le shell courant par la commande
- `exit [n]` : termine le shell avec un code de retour égal à `n`
- `export [-nf] [name[=word]] …`
  `export -p` : les noms spécifiés sont exportés
- `fc [-e ename] [-nlr] [first] [last]`
  `fc -s [pat=rep] [cmd]` : affiche ou manipule les commandes mémorisées par le mécanisme d'historique
- `fg [jobspec]` : place jobspec en premier plan

Les fonctions internes de bash (4/7)

- `getopts optstring name [args]` : permet l'étude des arguments de la ligne de commande
- `hash [-r] [name]` : manipule la zone mémorisant le chemin des commandes
- `help [pattern]` : aide sur les fonctions internes
- `history [n]`

  `history -rwan [filename]` : affiche ou manipule le fichier historique
- `jobs [-lnp] [jobspec …]`

  `jobs -x command [args …]` : affiche ou manipule les travaux
- `kill [-s sigspec | -sigspec] [pid | jobspec] …`

  `kill -l [signum]` : émission d'un signal à destination d'un processus

Les fonctions internes de bash (5/7)

- `let arg [arg …]` : évaluation d'expressions arithmétiques
- `local [name[=value] …]` : création de variables locales
- `logout` : quitte un login shell
- `popd [+/-n]` : dépile un répertoire de la pile des répertoires
- `pushd [dir]`

  `pushd +/-n` : empile un répertoire sur la pile des répertoires
- `pwd` : affiche le répertoire de travail courant
- `read [-r] [name …]` : lecture d'une ligne à partir de l'entrée standard
- `readonly [-f] [name …]`

  `readonly -p` : l'attribut lecture seule est fixé pour les noms spécifiés

Les fonctions internes de bash (6/7)

- `return [n]` : termine la fonction avec la valeur de retour `n`
- `set [options] [-o\_option] [arg …]` : manipulation des indicateurs et des variables du shell
- `shift [n]` : décalage de `n` positions vers la gauche des arguments de la ligne de commande
- `suspend [-f]` : suspend l'exécution du shell jusqu'à la réception du signal `SIGCONT`
- `test expr`

  `[ expr ]` : retourne l'évaluation de l'expression conditionnelle `expr` (0 : vrai, 1 : faux)
- `times` : affiche les temps d'exécution « utilisateur » et « système » du shell et des commandes lancées à partir de celui-ci

Les fonctions internes de bash (7/7)

- `trap [-l] [arg] [sigspec]` : définit la commande associée à la réception d'un signal
- `type [options] name [name ...]` : renseigne sur le type de de la commande `name` (alias, exécutable, fonction, fonction interne, mot réservé)
- `ulimit [-SHacdfmstpnuv [limit]]` : fixe les limites (soft et hard) du shell et de ses processus
- `umask [-S] [mode]` : fixe les droits par défaut des fichiers créés
- `unalias [-a] [name ...]` : retire un alias
- `unset [-fv] [name ...]` : annule la définition d'une variable
- `wait [n]` : attend la terminaison d'un processus et retourne son code de retour

Les mots clefs, les alias, les fonctions et les commandes

- Ces 4 objets manipulés par le shell ont en commun de toujours se trouver en début de ligne de commande
- Dans certains cas, il peut y avoir des alias, des fonctions et/ou des commandes de même nom
- Pour sélectionner l'un ou l'autre, le shell applique l'algorithme suivant :

```
1  si shell interactif et si premier mot de la commande = un alias alors
2     effectuer la substition
3  finsi

4  si le premier mot = un mot clef alors
5     continuer l'analyse de la commande
6  sinon
7     si le premier mot de la commande = une fonction alors
8        exécuter la fonction
9     sinon
0        exécuter la commande
1     finsi
2  finsi
```

Les mots clefs, les alias, les fonctions et les commandes (suite)

Remarques :

- Les fonctions internes `builtin`, `command` et `enabled` influent sur l'algorithme précédent
- La commande `type` permet d'obtenir des informations sur un objet

Programmation modulaire en shell

- Il est possible de décomposer les « gros » scripts en plusieurs fichiers grace à la fonction `source`
- La fonction `source` (que l'on peut abréger en `.`) permet d'exécuter, dans l'environnement courant, le fichier shell passé en argument
  - Les instructions présentes dans le fichier seront exécutées
  - Les fonctions et/ou les variables présentent dans le fichier deviendront donc connues du programme
- Par défaut la variable d'environnement `PATH` est utilisée pour trouver le fichier si son nom ne contient pas de `/`

Exemple d'utilisation de source

- Fichier « bibiothèques de fonctions » : `bib.sh`

```
1  #!/bin/sh

2  f() {
3      echo "Fonction f : ok"
4  }

5  a=2
```

- Script utilisant la bibliothèque de fonctions :

```
1  #!/bin/sh

2  # Chargement des fonctions
3  . ./bib.sh

4  # tests
5  f
6  echo "a = $a"
```