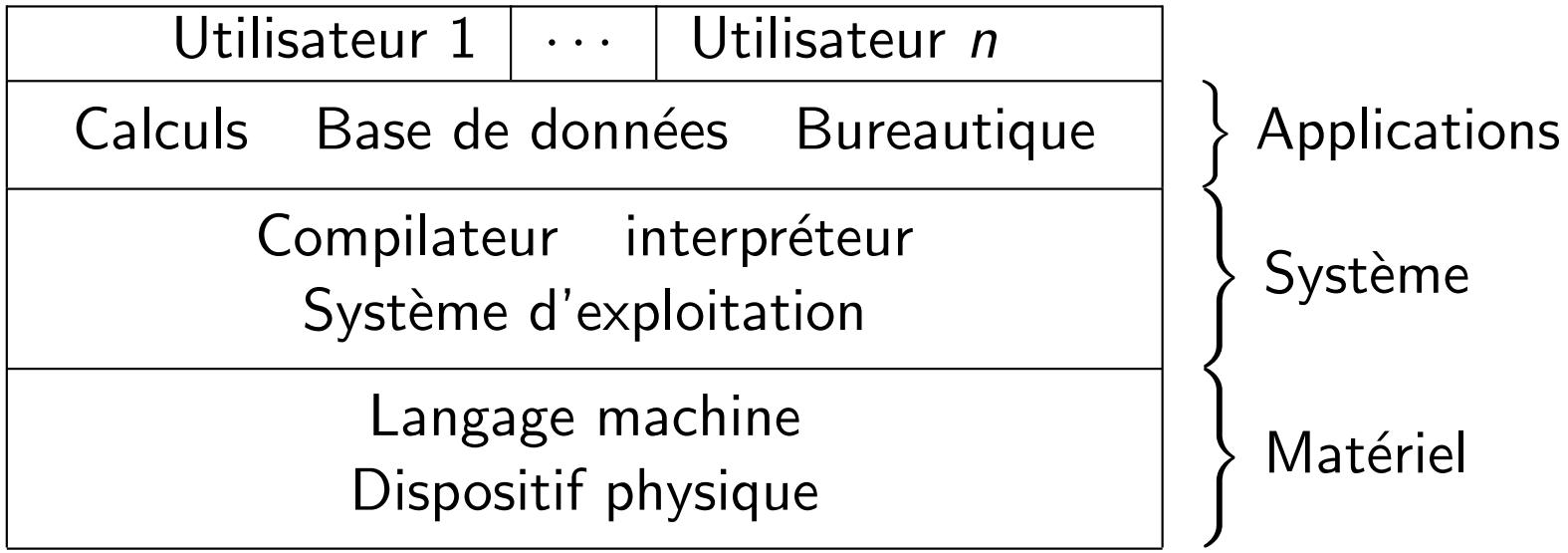
Shell(s) et langage(s) de script
Premiers pas

Formation continue — Université Lille 1
Pour toutes remarques : Alexandre.Sedoglavic@univ-lille1.fr

Licence trimestre 3 — 2008-09

Très schématiquement (et artificiellement), on peut répartir les *processus* en deux grandes catégories :

- les processus **d'applications** qui accomplissent des tâches souhaitées par l'utilisateur (calculs scientifique, base de données, bureautique, etc.) ;
- les processus **systèmes** qui permettent l'exploitation des ressources de l'ordinateur (processeurs, mémoire, terminaux, clavier, disques, coexistence/communication de plusieurs applications, etc).



Le système d'exploitation permet aux applications :

- d'utiliser les ressources matérielles de la machine ;
  - d'ordonner leurs exécutions les unes par rapport aux autres (éviter l'occupation du processeur par une application endormie, définir des priorités),
  - de gérer des droits (exécution, lecture) à des fin de sécurisation ;
- de communiquer :
  - par l'intermédiaire de la mémoire vive,
  - par l'intermédiaire de la mémoire persistente (disque),
  - par des structures had hoc (files de messages, sémaphore pour la synchronisation, etc).

Dans cette optique toute tâche complexe impliquant plusieurs applications doit être codée et prévue en bas niveau (langage C par exemple) en utilisant la connaissance du système (2 cours à venir).

Comment sans cela permettre à l'utilisateur d'utiliser les applications mises à sa disposition en les "combinant" au grès de sa fantaisie et de ses besoins ?

La façon la plus simple (et primitive) d'envisager la communication entre applications est de considérer ces dernières comme des *filtres*. Comme tout processus, un filtre possède (au minimum) 3 fichiers d'entrée-sortie :

- 0 stdin est l'entrée standard (par défaut, le clavier) ;
- 1 stdout est la sortie standard (par défaut, l'écran) ;
- 2 stderr est la sortie des erreurs (par défaut, l'écran).

De plus, chaque processus retourne à son père (son processus créateur) un octet qui est un *code de retour*.

Un filtre est une fonction invoquée par un identifiant (ls), des options (-al), des arguments (`/bin`) qui prend en paramètre une suite d'octets depuis l'entrée standard, retourne des octets dans ses sorties et produit un code de retour.

Une fonction a un *effet latéral* si elle modifie un état autre que ses valeurs de retour. Pour être utile, les filtres ont des effets latéraux divers (affichage, saisie, connexion à un serveur, création/modification/destruction de fichiers, etc).

Des processus particuliers : les shells

Un interpréteur de commandes :

- ne fait pas partie du système d'exploitation (c'est un processus comme les autres qui l'utilise) ;
- est une interface (texte) entre l'utilisateur, les applications disponibles et l'OS ;
- permet de faire des appels à l'OS depuis un interpréteur sans avoir besoin d'écrire du code C ;
- permet d'écrire des suites d'instructions réalisant une tâche complexe en manipulant des filtres.

Ce type de processus est appelé *shell* par analogie avec une *coquille* qui sert d'interface avec le *noyau* du système.

Comme le shell est un interpréteur, les suites d'instructions ne sont pas compilées et donc sont portables sur tout UNIX.

En résumé, un shell permet

- de lancer un processus considéré comme un filtre ;
- la combinaison (redirection, tube) de filtres et donc du traitements de l'information et des effets latéraux associés ;
- la programmation de tels combinaison avec des structures de contrôle (itération, conditionnelle) et des variables ;

mais aussi

- d'envoyer des signaux aux processus (détruire des applications) ;
- de fournir en héritage un contexte (ensemble de variables) à un processus lancé par le shell.

La philosophie générale peut être résumée par les maximes suivantes :

1. Small is beautiful ;
2. Make each program do one thing well ;
3. Build a prototype as soon as possible ;
4. Choose portability over efficiency ;
5. Store data in flat text files ;
6. Use software leverage to your advantage ;
7. Use shell scripts to increase leverage and portability ;
8. Avoid captive user interfaces ;
9. Make every program a filter.

Quelques critiques sur les shells

Dans *The UNIX-haters handbook* (disponible sur le web), on trouve les critiques suivantes sur les shells :

- le développement c'est fait par accrétion anarchique sur 30 ans (la commande *externe* ls qui liste les fichiers d'un répertoire a plus de 18 options, plusieurs syntaxes d'*expressions régulières* cohabitent, etc.) ;
- la gestion des erreurs n'est, en générale, pas correctement faîtes par les programmeurs ;
- ils ne sont pas standardisés i.e. il existe plusieurs shell avec des syntaxes différentes et incompatibles ;
- les suites d'octets constituant le flux d'informations entre applications ne sont pas typées ;
- ils nécessitent un apprentissage (et ne sont donc pas destinés à la majorité des utilisateurs) car leurs syntaxes et leurs grammaires ne sont pas intuitives.

Il existe plusieurs interpréteurs de commandes qui sont :

- dérivés du Bourne shell (sh, AT&T, 1977) comme ksh (korn shell), bash (Bourne again shell), zsh (zero shell), etc. ;
- dérivés du C shell (csh, BSD, 1979) comme tcsh (Tenex C shell), etc.

Une standardisation basée sur le bash est en cours : nous allons la suivre (autant que faire ce peu) dans ce cours.

Le shell peut être utilisé suivant 2 modes :

1. En mode interactif, il permet l'exécution de processus tout en offrant des *opérateurs* combinant leurs actions et un langage de programmation (cf. cours prochain) ;
2. En mode *batch*, les programmes associés à ce langage sont définis dans un *script* que le shell interprète (pas de compilation).

Nous allons présenter succinctement quelques notions annexes utiles.

Représentation de l'utilisateur par le système

Tout utilisateur — considéré comme une entité connue par le système d'exploitation — est caractérisé par

- son *login* i.e. le nom d'utilisateur ;
- son mot de passe ;
- un unique numéro d'identification (uid) ;
- un numéro de groupe d'utilisateur (guid) auquel il appartient ;
- un nom d'usage ;
- un répertoire i.e. son espace disque ($HOME) ;
- un interpréteur shell.

Par exemple pour le superutilisateur, on trouve les informations suivantes dans le fichier `/etc/passwd` :

```
root:x:0:0:root:/root:/bin/bash
```

Fichier et informations utilisateur relatives

Un fichier est l'abstraction d'un flux linéaire d'octets.

Aucune information sur l'organisation de l'espace du support à ce niveau d'abstraction. Pour manipuler les fichiers, il faut pouvoir les identifier par leurs caractéristiques :

- nom, type, taille du fichier ;
- propriétaire du fichier, son groupe ;
- date de création, date de dernière modification ;
- protection : qui a droit de le lire et de le manipuler.

Au fichier foo.bar sont associées les informations

```
-rw-r--r--    1 sedoglav calforme 0 Aug 19 05:09 foo.bar
```

Ces informations correspondent dans l'ordre aux droits, nombre de liens, au propriétaire, à son groupe, à la taille, à la date de création et au nom du fichier.

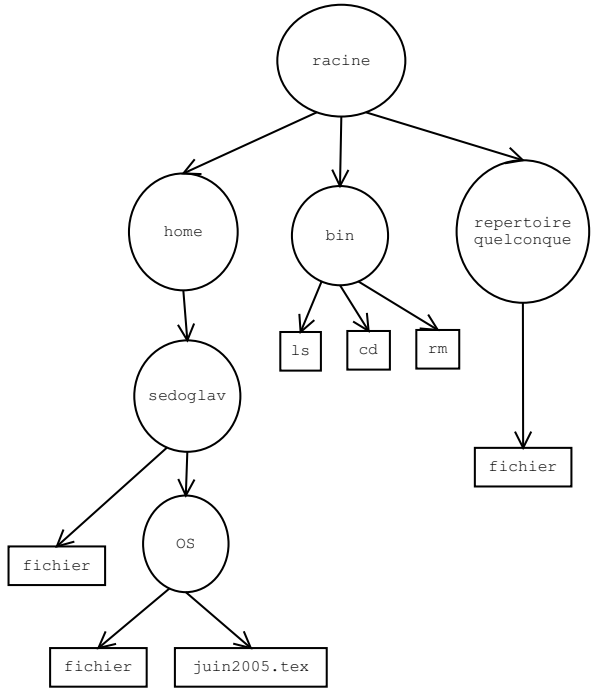
Organisation arborescente des fichiers

La communauté des fichiers est organisée en arbre i.e. en un ensemble de nœuds reliés par des arêtes orientées : chaque nœud a exactement une arête pointant vers lui (à l'exception de la racine qui est un nœud sans prédécesseur). Les feuilles sont les noeuds sans successeur.

Les feuilles sont des fichiers et les nœuds sont des *répertoires*. On peut ainsi définir un chemin d'accès à un fichier :

- absolu : depuis la racine ;
- relatif : répertoire courant.

Le fichier `juin2005.tex` est localisé par le chemin d'accès



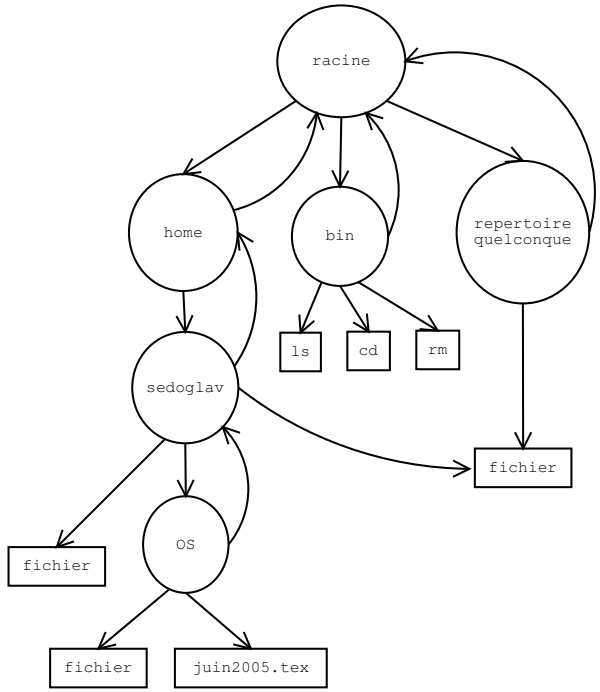

```
racine -> home -> sedoglav -> OS ->
```

Les répertoires sont des fichiers contenant l'information liée à leurs arêtes et constituant le graphe.

Organisation en graphe

Cette représentation est assoupli en autorisant plusieurs arêtes à pointer sur un même élément et en faisant pointer chaque répertoire sur son prédécesseur. On obtient ainsi un graphe permettant :

- de remonter l'arborescence depuis n'importe quel répertoire sans repartir de la racine ;
- de permettre l'accès depuis le répertoire `sedoglav` à un fichier référencé dans un autre répertoire (lien symbolique codé par un fichier).



Les répertoires étant des fichiers, ils ont les mêmes attributs (propriétaire, droits, etc).

Dans les OS dérivés d'UNIX, le codage des droits se fait sur 10 bits qui sont dans l'ordre :

- le type du fichier (`d` pour répertoire, `l` pour un lien, `c` et `b` pour un périphérique, `p` pour un tube, `-` pour un fichier classique) ;
- **r** le fichier est lisible par le propriétaire (- dans le cas contraire) ;
- **w** le fichier est modifiable par le propriétaire (- sinon) ;
- **x** le fichier est exécutable par le propriétaire (- sinon).

Le groupe suivant de 3 bits reprend le même principe mais définit les droits pour les membres du groupe auquel appartient le propriétaire.
Le dernier groupe reprend le même principe mais concernant les autres utilisateurs.
Ainsi le fichier `foo.bar` n'est pas un répertoire, il n'est exécutable par personne, il est lisible par tout le monde et n'est modifiable que par son propriétaire.

Remarques sur ce type d'abstraction

Dans les interfaces graphiques, on présente souvent les répertoires suivant la métaphore d'un dossier *contenant* les fichiers. Il est important de distinguer la métaphore de l'abstraction. La taille d'un répertoire n'est pas celle des fichiers qu'il *structure* mais celle nécessaire pour coder l'ensemble des liens.

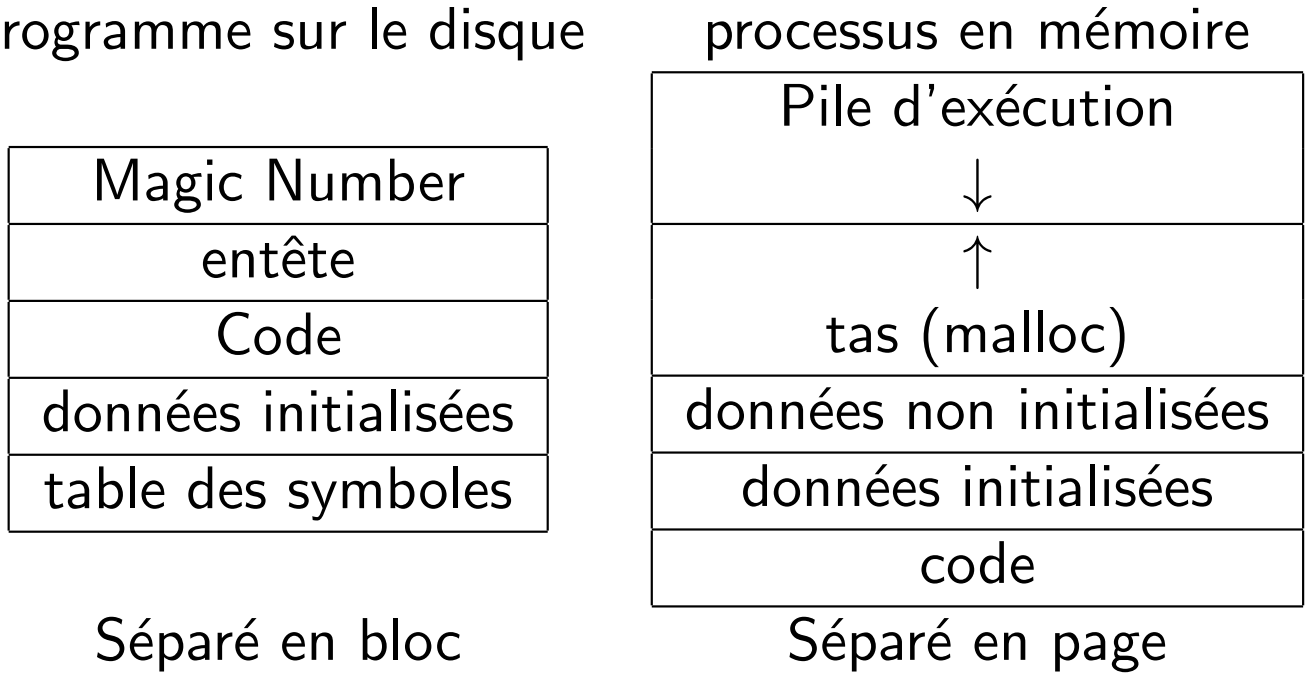
L'OS permet l'accès à des fichiers stockés sur des supports distincts de l'ordinateur local (cf. la notion de montage) par le biais de répertoire.

Il existe des fichiers *spéciaux* ; par exemple `/dev/null` est un fichier détruisant tout octet que l'on écrit dedans.

Les droits d'exécution ne définissent pas un exécutable (les premiers octets d'un fichier indiquent si c'est un script, un object ou un exécutable) ; e.g. les répertoires doivent être exécutables pour être parcourables.

Notion de processus

Un processus est l'abstraction d'un programme exécuté par la machine.



Comme exemple de *magic number*, signalons qu'un fichier commençant par `#!` est sensé être un script pour un interpréteur (ces 2 caractères sont suivis par le chemin d'accès à l'interpréteur).

Chaque processus est représenté dans l'OS par un *bloc de contrôle de processus* contenant les informations suivantes :

- l'état du processus (nouveau, prêt, en cours d'exécution, en attente, arrêté, etc.) ;
- les numéros associés au processus ;
- l'état des registres du microprocesseur associés au processus et notamment :
  - les différents numéros de segments (mémoire, code, pile, etc.),
  - un pointeur sur la prochaine instruction à exécuter,
  - les registres à usage généraliste ;
- la liste des fichiers ouverts et l'ensemble des informations associées aux entrées-sorties ;
- des statistiques sur l'utilisation des ressources de la machine (temps d'occupation du processeur, etc.) ;
- les limites maximales de la mémoire (début et taille de ces espaces).

Les informations attachées à un processus

L'utilisateur dispose de renseignement sur un processus :

- le pid i.e. numéro identificateur du processus ;
- le ppid i.e. numéro identificateur du processus père ;
- l'uid i.e. numéro identificateur du propriétaire du processus ;
- le numéro du groupe auquel appartenait le propriétaire du processus (il peut appartenir à plusieurs groupes) ;
- l'heure et la date de création du processus ;
- des statistiques sur les ressources (processeur, etc.) utilisées ;
- la taille mémoire utilisée par le processus (en distinguant chaque composante : code, données, etc).

Mais aussi,

- un masque indiquant la sensibilité aux signaux ;
- son état (en cours d'exécution, zombie, arrêté, etc.) ;
- la priorité d'exécution ;

qui sont des notions liées à l'ordonnancement.

Interface interactive du shell

Le shell permet notamment de manipuler les abstractions précédentes. Pour ce faire, le shell propose une *invite de commande* que nous désignerons par %.

1. Cette invite est associée à un éditeur en ligne (readline) et à l'ensemble de ses possibilités (déplacement, copié collé, etc).
2. Un backslash (code ASCII 92) suivi d'un retour chariot permet d'éditer une commande sur plusieurs lignes.
3. Un caractère dièse (code ASCII 35) débute un commentaire.

```
% # ceci est un commentaire \
>  \
> # encore un commentaire avec \ au milieu
%
```

Les commandes shell sont de 2 types : interne et externe.

Un shell permet d'exécuter depuis une mémoire permanente une commande externe :

```
% <commande externe> [option(s)] [argument(s)]
```

i.e. un programme exécutable ; le shell se *clone* puis se *mute* en un processus associé à l'exécutable.

Par exemple, l'exécutable de la commande `ls` se trouve dans le répertoire `/bin` ; il affiche les informations relatives à un fichier :

```
% /bin/ls -l /usr/bin/man
-rwxr-xr-x  1 root root 46308 Apr  8  2005 /usr/bin/man
```

- `/usr/bin/man` est un argument indiquant que l'on désire un affichage concernant ce fichier ;
- `-l` est une option indiquant que l'on désire un affichage de toutes les informations.

Le manuel d'utilisation man est un outil fondamental ; une première chose à faire est de lire l'aide sur le manuel en utilisant la commande externe % man man dans votre interpréteur de commandes favori.

- `% man -k mount` affiche la liste des pages d'aide contenant le mot mount. Entre autre :

  ```
  mount    (2)  - mount and unmount filesystems
  mount    (8)  - mount a file system
  ```

  `% man -a mount` affiche successivement l'ensemble de ces pages d'aide ;
- % man -S8 mount affiche l'aide sur mount issue de la section 8 du manuel.

On peut aussi utiliser l'utilitaire info mais, bien que plus évolué (liens hypertext), il n'est pas forcement complet.

Le cours pourrait (en théorie) se faire uniquement à partir du man. Il décrit non seulement les filtres mais aussi toutes les notions de bases nécessaires à la compréhension du système.

Chaque filtre doit/devrait avoir ces propres pages d'aide qui sont structurées suivant les sections :

- NAME : le nom du filtre et une description sommaire de son action ;
- SYNOPSIS : un résumé de la syntaxe de l'appel du filtre avec les options et les arguments ;
- DESCRIPTION : la description complète de l'action du filtre ;
- OPTIONS : la suite des options disponibles et la façon dont elles modifient l'action du filtre ;
- EXAMPLES : un ou plusieurs exemples d'utilisation ;
- SEE ALSO : des pages d'aides ayant rapport avec le filtre.

Des sections EXIT STATUS, STDIN, STDOUT et STDERR devraient — selon la norme — décrire les codes de retour et la gestion de ces fichiers par le filtre.

Une commande interne est un filtre implanté dans le shell et ne correspond (en théorie) à aucun fichier exécutable.
L'objectif étant :

- d'augmenter les performences de filtres très fréquement utilisés ;
- de permettre des fonctionnalités difficiles à mettre en œuvre avec un code indépendant du shell.

Dans le GNU-bash-3.0, les commandes false, true, kill, pwd et newgrp sont externes bien que la norme les considère comme internes.

La commande interne type retourne dans STDOUT des informations sur les commandes (sont elles internes, externes, etc).

La commande externe which foo retourne dans STDOUT le chemin d'accès à la commande externe foo si elle le trouve.

Commandes internes spéciales

Les commandes internes suivantes

```
break, colon, continue, dot, eval, exec, exit, export,
readonly, return, set, shift, times, trap, unset
```

sont qualifiées de *spéciales* car :

- une erreur de syntaxe dans leurs usages peut causer la destruction du shell ;
- l'affectation des variables (voir plus loin) au cours de l'exécution de ces commandes reste valide après leurs terminaisons.

Ce n'est pas le cas des autres commandes (internes ou externes).

Les autres commandes internes sont :

```
alias, bg, cd, command, false, fc, fg, getopts, jobs,
kill, newgrp, pwd, read, true, umask, unalias, wait
```

L'environnement d'exécution du shell est composé des informations suivantes :

- les fichiers fournis à l'invocation du shell et ceux produits par l'exécution de la commande en cours ;
- le répertoire courant (définit par la commande cd) ;
- un masque de création du fichier i.e. les droits par défaut des nouveaux fichiers (spécifiés par la commande umask) ;
- la gestion des interruptions (définie par la commande interne trap) ;
- les options du shell — comme l'éditeur par défaut (spécifié par la commande interne set) ;
- les variables du shell (cf. section suivantes) ;

nous verrons dans un prochain cours :

- les fonctions shell définies par l'utilisateur ;
- les alias i.e. évaluations et substitutions de textes.

Le shell admet les opérateurs suivants :

espace, tab	séparateurs ;
enter	envoit d'une expression (instruction) ;
\	caractère d'échappement ;
|	concaténation d'entrées-sorties de 2 filtres ;
&	lance un processus en arrière fond,
	utilisé aussi dans les redirections ;
;	séparateur entre 2 expressions ;
()	groupement et calcul ;
<	redirection d'entrée ;
>	redirection de sortie ;
||	ou logique entre expression ;
&&	et logique entre expression ;
;;	fin d'un case.

Dans un cours suivant, nous étudierons la syntaxe et la grammaire du shell induite par les commandes et les opérateurs i.e. comment les commandes et les opérateurs produisent des expressions. Pour l'instant, nous allons décrire quelques opérateurs courants.

Interprétation séquentielle vs asynchrone

- subexpr1 <optionnel> ; subexpr2 ; . . . < \optionnel>
  - l'opérateur ; permet de séparer l'exécution de commandes %cd / ; ls ;. Par défaut, les shells attendent la fin de l'exécution d'une commande avant de permettre la saisie et l'exécution d'une autre ;
  - le code de retour de l'expression est celui de la dernière sous-expression dans la liste.
- subexpr1 & <optionnel>subexpr2 & . . . < \optionnel>
  - les shells permettent aussi de lancer une application en *tâche de fond* (processus clone du bash muté en l'application) et ainsi l'exécution d'une autre (même si la première n'est pas terminée). Pour ce faire, on termine l'expression par & ;
  - si STDIN n'est pas précisé et que subexpr1 n'est pas interactive, l'entrée standard est `/dev/null` ;
  - le code de retour d'une expression asynchrone est 0 dans tous les cas.

Pour détruire un processus dont le shell attend la terminaison, on utilise le raccourci clavier CTRL-C.
Pour interrompre sans détruire un processus, on utilise le raccourci clavier CTRL-Z. Pour le relancer :

- en tâche de fond, on utilise la commande interne `bg` ;
- en avant plan, on utilise la commande interne `fg`.

La commande externe `ps` retourne dans STDOUT les informations associées aux processus.

```
% ps -l
F S UID  PID PPID C PRI NI ADDR SZ WCHAN TTY     TIME CMD
0 S 613 2434 2426 0 75 0 -  954 rt_sig pts/1 00:00:00 bash
```

La commande externe `kill -<Signal> <PID>` envoit un signal au processus d'identificateur `PID`. Les principaux signaux sont :

Signal	Signification
15	terminaison de processus
9	destruction inconditionnelle de processus (CTRL-C)
19	suspension de processus (CTRL-Z)
18	reprise d'exécution d'un processus suspendu

Une expression entre parenthèses est interprétée par un shell-fils du shell courant et pas par ce dernier :

```
% ( exit ) # est bien diff\'erent de exit
```

On dispose de 2 opérateurs conditionnels :

- `subexpr1 && subexpr2` : subexpr2 est exécuter si, et seulement si, subexpr1 retourne 0 ;
- `subexpr1 || subexpr2` : subexpr2 est exécuter si, et seulement si, subexpr1 retourne un code non nul.

Ces deux règles sont appliquées par le shell lorsqu'une suite de commandes contient plusieurs opérateurs `&&` et `||`. Ces deux opérateurs ont la même priorité et leurs évaluations s'effectue de gauche à droite.

Le code de retour des expressions ainsi construites est le code de retour de la dernière sous-expression exécutée.

Les filtres sont associées à des flux d'octets depuis le fichier standard STDIN vers les fichiers standards STDOUT et STDERR. Ces flux peuvent être redirigés par les opérateurs :

- $n >$foo : fichier standard de descripteur $n(= 1, 2)$ dans le fichier foo (création ou écrasement) ;
- $n >>$foo : fichier standard de descripteur $n(= 1, 2)$ dans le fichier foo (création ou ajout) ;
- $n <$foo : le fichier foo est envoyé dans le fichier de descripteur $n(= 0, 1, 2)$;
- $n <<$ EOF (texte EOF) : insertion de texte dans le fichier de descripteur $n(= 0, 1, 2)$;
- $|$: tube de communication entre 2 filtres ;
- $n >$&m : réoriente le flux de sortie du fichier de descripteur n dans celui de descripteur m ;
- $n <$&m : réoriente le flux d'entrée du fichier de descripteur n dans celui de descripteur m.

Si l'entier n est omis, la redirection concerne STDOUT pour les sorties et STDIN pour les entrées.

Quelques illustrations des redirections

```
% ls /bin 1> /tmp/foo ; grep sh 0< /tmp/foo # correct
% ls /ntn /bin 1>/dev/null 2> /tmp/err # correct
% grep sh 0< ls # incorrect car il n'y pas pas de fichier ls
% ls 1> grep sh # incorrect car cr\'ee le fichier grep
% (ls /ntn /bin 2>&1) 1>/tmp/foo# manipule 2 filtres, ls et sh
```

Les commandes suivantes sont équivalentes :

```
% ls /bin>/tmp/foo;grep sh</tmp/foo>&result;
% ls /bin | grep sh >& result # >& redirige stdout et stderr
```

Un exemple d'insertion de texte où le filtre grep prend son entrée depuis le clavier jusqu'à la saisie de pourfinir :

```
% grep tata << pourfinir
? abcd
? abcdtata
? pourfinir
abcdtata
```

Les pipelines (tubes de communication)

Un tube est une suite d'une ou plusieurs expressions séparée par l'opérateur | :

- <optionnel> !< \optionnel> subexpr1 <optionnel>| subexpr2 . . . < \optionnel>
  La sortie standard de tous — sauf le dernier – les filtres associés aux sous-expressions est redirigée vers l'entrée standard du suivant ;
- l'opérateur | est prioritaire sur les autres redirections ;
- si le pipeline n'est pas lancé en tâche de fond, le shell attend la fin de la dernière commande du pipe avant de rendre l'invite de commande ;
- le code de retour de l'expression et celui de la dernière commande du pipe.

Dans ce cas, l'opérateur ! est une négation du code de retour i.e. $!0 = 1$ et si $n \neq 0$ alors $!n = 0$.
Le ! est aussi utilisé par la commande interne history (cf. travaux pratiques et % man history).

Dans un processus, les fichiers accessibles sont associées à des entiers (les descripteurs de fichier). Un exemple ad hoc :

```
#include <unistd.h>
#define DESCRIPTEUR 8
#define NBCAR 3
int main(void){
    char tab[NBCAR] = { 'a', 'b', '\n' } ;
    return write(DESCRIPTEUR, tab, NBCAR);
}
```

pour montrer que l'on peut rediriger d'autres fichiers que les fichiers standards :

```
% ./a.out 8> /tmp/foo ; cat /tmp/foo
ab
%
```

Attention, pour utiliser un fichier il faut lui associer un descripteur par un appel (open) à l'OS (ce n'est pas fait dans le code ci-dessus). Plus de détails dans le cours "Pratique des systèmes".

Un exemple acrobatique

Une fois compilé et exécuté, le code suivant produit un processus qui lit NBCAR octets depuis sa sortie standard et les écrit sur son entrée standard :

```
#include <unistd.h>
#define STDOUT 1
#define STDIN 0
#define NBCAR 2
int main(void){
        char tab[NBCAR] ;
        read(STDOUT, tab, NBCAR) ;
        return write(STDIN, tab, NBCAR);
}
```

Il permet d'illustrer les redirections suivantes :

```
% echo "ab" > /tmp/foo ; ./a.out 1< /tmp/foo 0> /tmp/bar ;\
> cat /tmp/bar
ab% # la commande echo retourne une ligne de texte a stdout
```

- un *paramètre* du shell peut être identifié par un nombre, un caractère spécial (cf. section suivantes) ou un nom (une chaîne de caractères alphanumérique qui n'est ni un nombre ni un caractère spécial) ;
- une *variable* du shell est un paramètre identifié par un nom ;
- un *paramètre de position* est un paramètre qui n'est ni spécial ni une variable.

Un paramètre est affecté s'il possède une *valeur* (null est une valeur).
Une variable ne peut être désaffectée que par la commande interne unset.

Paramètres spéciaux et de position

- `0` : le nom de la commande en cours ;
- `#` : son nombre de paramètres de position ;
- `*`, `@` : tous ses paramètres de position ;
- `1` à `9` : ses 9 premiers paramètres de position ;
- `x` : le paramètre de position $x (> 9)$;
- `$` : le pid de la commande courante ;
- `_` : le dernier paramètre manipulé (non normalisé) ;
- `-` : les drapeaux (options) de la commande courante ;
- `?` : toutes les commandes ont un code de retour — codé sur un octet — (*exit-status*) i.e. une valeur entière qui fournie une information sur le déroulement de la dernière commande exécutée.
  - déroulement normal $\Rightarrow$ `?` $= 0$,
  - déroulement anormal $\Rightarrow$ `?` $\neq 0$;

  Nous verrons en C comment renvoyer le code de retour ;
- `$!` : le pid du dernier processus lancé en arrière fond.

Les paramètres spéciaux peuvent s'utiliser tels quels dans un shell :

```
% false ; echo $0 $$ ; ps | grep bash
1 bash 2977
2977 pts/3    00:00:00 bash
```

En mode interactif, on les affecte avec la commande interne set :

```
% echo $# # nous verrons plus tard le sens du $
0
% set foo bar ; echo $# $1 $2
2 foo bar
```

La commande interne `shift` permet le décalage des paramètres numérotés (`1` est perdu et `#` est mis à jour).

```
% shift ; echo $# $@
1 bar
```

Le shell dispose de variables que la commande interne set permet d'afficher :

```
% set
USER=sedoglav
LOGNAME=sedoglav
HOME=/home/enseign/sedoglav
PATH=/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
MAIL=/var/mail/sedoglav
SHELL=/bin/csh
HOSTTYPE=i586-linux
PWD=/home/enseign/sedoglav
GROUP=enseign
LANG=fr_FR
SYSFONT=lat0-16
TMP=/home/enseign/sedoglav/tmp
HOSTNAME=lxt2
```

La commande set permet aussi de manipuler les options du shell. Par exemple, %set -o vi permet de passer en mode d'édition vi.

Les variables

Définition et affectation : une variable est définie dès qu'elle est affectée. En sh, `% FOO="Bonjour le monde"`. En csh,

```
% set FOO="Bonjour le monde"
```

La commande interne `echo` permet d'afficher l'argument qui lui est fourni :

```
% echo FOO
FOO
```

Pour évaluer une variable, il faut préfixer son nom par $.

```
% echo $FOO
Bonjour le monde
```

En sh, la commande interne `export` étend la porté d'une variable : par défaut, cette dernière n'est connue que par le processus courant ; après coup, cette variable est connue par tous les processus fils de ce dernier. En csh, on utilise :

```
% setenv FOO "Bonjour le monde"
```

Quelques variables d'environnement

Les variables définies dans les fichiers `/etc/profile` et `~/profile` sont créées lors de l'ouverture d'une session.

PATH	les répertoires dans lesquels sont cherchés les exécutables des commandes externes
HOME	votre répertoire de travail
TERM	le type de terminal
PWD	le répertoire courant
DISPLAY	cette variable est utilisée par l'interface graphique pour savoir où se fait l'affichage
PS1	l'invite de commande

Ces variables d'environnement peuvent être utilisées depuis un programme C (fonction getenv) lancé depuis le shell.

Dans un shell, **tout n'est que chaîne de caractères**.

Chaque commande est une chaîne que le shell évalue. On peut influer sur cette évaluation grâce aux délimiteurs suivants :

- les quotes ' ' bloquent l'évaluation ;
- les guillemets " " forment une chaîne après évaluation des composantes ;
- les backquotes ` ` forment une chaîne évaluée comme une commande.

```
% echo '$FOO'
$FOO
% echo "echo '$FOO'"
echo 'Bonjour le monde'
% set BAR="n\'importe quoi" ; echo $BAR
n\'importe quoi
% set BAR=`n\'importe quoi`
n'importe: Command not found.
```

Première utilisation d'un script

Les commandes shell peuvent être placées dans un fichier `foo` puis on peut soit :

- demander au shell courant l'interprétation de ce fichier par la commande `. foo` ;
- rendre exécutable ce fichier par la commande externe %chmod u+x foo et l'exécuter directement i.e. les commande sont interprétées dans un shell-fils.

Un *script* est un fichier contenant des commandes shell.

Par convention et lorsqu'on l'on souhaite avoir un script exécutable, la première ligne indique l'interpréteur à utiliser (`#!/bin/sh` par exemple).

Les paramètres spéciaux et de position sont particulièrement utiles dans l'écriture de scripts interprétables.

```
% ls /repnonexistant ; echo $?
ls: /repnonexistant: No such file or directory
1
% cat monscript
#!/bin/bash
echo $0 1>&2 ; shift ; ls $@
% ./monscript /repnonexistant /bin 1> /tmp/foo ; \
>  echo "retour : " $? ; grep sh /tmp/foo
./monscript
retour : 0
bash csh ksh sh tcsh zsh
```

Explicitons la distinction entre interprétation des commandes d'un script et son exécution :

```
% ./monscript /ntnt /bin >& /dev/null
./monscript
% . monscript /ntnt /bin >& /dev/null
bash
```

Les *expressions régulières* décrivent des propriétés de construction de chaînes de caractères. Pour ce faire, on utilise en shell les *métacaractères* :

- le point d'interrogation ? correspond à n'importe quel caractère (sauf EOL). L'expression régulière `b?l` représente les chaînes *bal* et *bol* et toutes les autres combinaisons comme *bwl* ;
- la paire de crochet `[ ]` permet de spécifier plus restrictivement un ensemble de caractères. L'expression régulière `dupon[dt]` ne représente que les chaînes *dupond* et *dupont*. L'expression `dupon[d-t]` représente les chaînes commençant par *dupon* et se terminant par une lettre comprise entre *d* et *t*. L'expression `dupon[^dt]` représente les chaînes commençant par *dupon* et ne se terminant ni par *d* ni par *t* ;
- l'étoile $*$ désigne $0, 1$ ou plusieurs caractères quelconques. Ainsi, $*$ représente toutes les chaînes.

Le préfixe \ (antislash) transforme un métacaractère en caractère.

Raccourci clavier et quelques caractères spéciaux utiles

La liste des raccourcis clavier est affichable par des commandes internes :

- `bind -p` pour bash ;
- `bindkey` pour csh.

Retenons pour mémoire :

CTRL-d	caractère fin de fichier
CTRL-\	stop la commande en cours

Pour approfondir l'usage d'un interpréteur de commande, la prochaîne étape consiste à étudier la syntaxe et la grammaire induite par les opérateurs et les commandes, l'évaluation associée, les expressions (simples et composées), les instructions de contrôle, les fonctions et le passage de paramètres, etc.