Université Grenoble Alpes
L3 MIAGE

Année 2025-2026

Cours
Introduction aux systèmes d'exploitation et à Unix

Laurence PIERRE
(Laurence.Pierre@univ-grenoble-alpes.fr)

I. Introduction

- Unix (Linux, étudié ici) est un **système d'exploitation**.

An operating system manages system resources by allocating and monitoring hardware and software resources for different tasks.

The operating system (OS) is a crucial component of a computer system, acting as the intermediary between the user and the hardware. It is responsible for managing the system's resources, which include the CPU, memory, disk space, and peripheral devices such as printers and scanners. The OS ensures that these resources are used efficiently and effectively, preventing conflicts and ensuring smooth operation.

One of the primary responsibilities of an OS is managing the CPU. The OS decides which processes get access to the CPU, and when. This is known as process scheduling. The OS uses algorithms to determine the order in which processes are executed, aiming to maximise CPU utilisation and minimise response time. The OS also manages the allocation of CPU time to different tasks, ensuring that all processes get a fair share of the CPU's resources.

Memory management is another crucial function of the OS. The OS keeps track of each byte in a system's memory and which processes are using which memory at any given time. When a process needs to be executed, the OS allocates memory for it. When the process is finished, the OS frees up the memory so it can be used by other processes. This is crucial for ensuring that the system's memory is used efficiently and that processes do not interfere with each other.

The OS also manages disk space. It keeps track of which files are stored where on the disk, and it manages the reading and writing of data to the disk. The OS also manages the allocation of disk space to different files and directories, ensuring that disk space is used efficiently and that files are easy to locate.

Finally, the OS manages peripheral devices. It communicates with these devices, sending them commands and receiving data from them. The OS also manages the allocation of these devices to different tasks, ensuring that they are used efficiently and that tasks do not interfere with each other.

https://www.tutorchase.com/answers/ib/computer-science/how-does-an-operating-system-manage-system-resources

Dans ce cours, nous nous concentrerons essentiellement sur la gestion des **fichiers**, et la vie/mort/communication des **processus**.

Note. Plusieurs variantes d'OS de type Unix existent ou ont existé, parmi elles **Linux** (et ses diverses distributions : Red Hat, Debian, Arch Linux,…) et **MacOS** (OS X) sur Mac.

- Au login, l'utilisateur est positionné dans son "répertoire de login" et un **interpréteur de commandes** est lancé (celui choisi par l'utilisateur ou l'administrateur au moment de la création du compte) :
  - sh (Bourne-shell),
  - bash (Bourne-again shell),
  - csh (C-shell, issu de Berkeley), ou tcsh,…

Shell

Most people know of shells as small protective coverings for certain animals, such as clams, crabs, and mollusks. You may also find a shell on the outside of an egg, which I highly recommend you remove before eating. In the computer science world, however, a shell is a software program that interprets commands from the user so that the operating system can understand them and perform the appropriate functions.

The shell is a command-line interface, which means it is soley text-based. The user can type commands to perform functions such as run programs, open and browse directories, and view processes that are currently running. Since the shell is only one layer above the operating system, you can perform operations that are not always possible using the graphical user interface (GUI). Some examples include moving files within the system folder and deleting files that are typically locked. The catch is, you need to know the correct syntax when typing the commands and you may still be prompted for a password in order to perform administrative functions.

http://techterms.com/definition/shell

- **Dans ce cours, nous verrons**
  - Unix du point de vue **utilisateur** (système de fichiers, commandes de base, redirections, config. du *shell*, gestion des processus,…),
  - puis du point de vue du programmeur : **programmation de certains de ces concepts** élémentaires, **en C** (création et communication des processus essentiellement).

*=> Il est **indispensable** de **maîtriser** la programmation C, voir partie mise à niveau…*

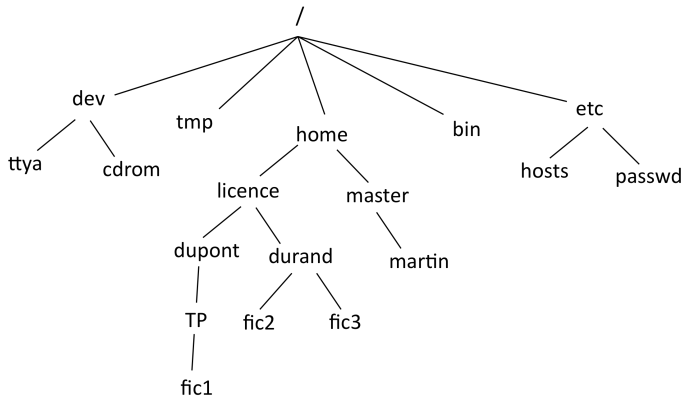
PARTIE 1. BASES UTILISATEUR

II. LE SYSTEME DE FICHIERS

II.1 Introduction

- La notion de **fichier** est un concept clé d'Unix. Noter que les périphériques apparaissent aussi comme des **fichiers**.
  Principalement trois grands types de fichiers :
  - fichiers ordinaires,
  - répertoires,
  - fichiers spéciaux (notamment pour les périphériques).

- Le système de fichiers est organisé en *structure arborescente* dont les *nœuds* sont des répertoires et les *feuilles* sont des fichiers ordinaires.
  Dans la suite nous illustrerons avec l'exemple fictif simple ci-dessous :



Note. En pratique le nom du répertoire de login d'un utilisateur peut être (et est généralement) identique au nom de l'utilisateur, mais rien ne l'impose. Dans les exemples ci-dessous, pour faciliter la distinction, nous

utiliserons des identifiants différents. Ainsi le nom du répertoire de login de l'utilisateur *ymartin* sera *martin*, le nom du répertoire de login de l'utilisateur *bdupont* sera *dupont*, et le nom du répertoire de login de l'utilisateur *mdurand* sera *durand*.

II.2 Désignation des fichiers

- Chemin absolu : liste des noms de répertoires jusqu'au fichier concerné, depuis la racine, séparés par des /

  Exemples : `/etc/passwd`
  `/home/licence/dupont`

  NB. Un chemin **absolu** commence toujours par /

- Chemin relatif (au répertoire courant) :

  Noter qu'on utilise **..** pour représenter le *répertoire père* (et **.** représente le *répertoire courant*).

  Exemples : en étant dans `/home/licence/dupont`

  - le fichier fic1 dans TP :
    `TP/fic1`
    (qui est aussi équivalent à `./TP/fic1`)
  - le fichier fic2 du répertoire durand :
    `../durand/fic2`
  - le répertoire martin dans master :
    `../../master/martin`

  NB. Un chemin **relatif** ne commence jamais par /

- Caractères autorisés dans un nom de fichier :
  Lettres majuscules et minuscules, chiffres, le point (.), le tiret (-) et le souligné (_).

- Interprétation des **caractères ? et *** :
  - le ? désigne n'importe quel caractère

  - le * désigne n'importe quelle suite de caractères (éventuellement vide)

  En outre, [...] permet de désigner un caractère quelconque appartenant à l'ensemble des caractères placés entre les crochets.

  Exemples :

  `du*` représente tout fichier dont le nom commence par "du"

  `fic?` représente tout fichier dont le nom est de la forme "fic" suivi d'un caractère

  `fic[0-9]` représente tout fichier dont le nom est de la forme "fic" suivi d'un caractère qui est un chiffre

  `[AB]xyz` représente tout fichier dont le nom commence par A ou B, suivi de "xyz"

- Interprétation du **caractère ~** : ***répertoire de login*** *d'un utilisateur*

  Exemples :

  - le répertoire martin de master (qui est ici le répertoire de login de l'utilisateur ymartin) :
    `~ymartin`
  - le fichier fic2 de durand (répertoire de login de l'utilisateur mdurand) :
    `~mdurand/fic2`

II.3 Manipulation des fichiers et répertoires

- *pwd* : donne le chemin absolu du répertoire courant.

  Exemple :
  ```
  $ pwd
  /home/licence/dupont
  ```

- *cd* : permet de se déplacer d'un répertoire à un autre répertoire (pourvu qu'on en ait l'autorisation).

  Exemple :
  ```
  $ cd ../durand
  $ pwd
  /home/licence/durand
  ```

cd sans argument permet de se replacer dans son répertoire "de login"

- *ls* : donne la liste des fichiers et répertoires (du répertoire courant ou du répertoire passé en paramètre).

Exemple :

```
$ ls
fic2      fic3
$ ls ..
dupont    durand
```

ls a de nombreuses options. Par exemple `ls -l` permet d'obtenir les informations de protections, propriétaire, taille, etc…

- *cat* ou *more* ou *less* : permettent de visualiser le contenu d'un fichier.

Exemple :

```
$ cat fic2
... affichage du contenu de fic2 ...

$ more fic2
... affichage du contenu de fic2 page par page ...
```
q : arrêt
blanc : page suivante
RC : ligne suivante

- *mkdir* : Permet de créer un nouveau répertoire.

- *rm* et *rmdir* : Permettent d'effacer un fichier (rm), ou un répertoire (rmdir) pourvu qu'il soit vide.

- *cp* : Permet de recopier un fichier.
  mv : Permet de renommer un fichier.

Exemple :

```
$ cp fic2 fic2bis

$ ls
fic2      fic2bis        fic3

$ mv fic2bis fic4

$ ls
fic2      fic3      fic4
```


II.4 Protection des fichiers

- Trois *groupes d'utilisateurs* :
  propriétaire ("**user**", u)
  membres du même groupe ("**group**", g)
  autres ("**others**", o)

- Trois *modes de protection* :
  autorisation **lecture** (r)
  autorisation **écriture** (w)
  autorisation **exécution** (x)

- On associe *à chaque groupe d'utilisateurs un mode de protection* (i.e. 3 attributs).

- Dans l'affichage fourni par *ls*, un 10ième attribut donne le type du fichier :
  d : répertoire,
  - : fichier ordinaire,
  b : fichier spécial de type bloc,
  c : fichier spécial de type caractère, etc…

Exemples :

```
$ ls -l /home/licence/durand/fic2
-rw-rw-r--  1 mdurand     1150 Sep 26 18:59 /home/licence.../fic2

$ ls -ld /home/licence/dupont/TP
drwxr-xr--  1 bdupont     512 Jul  5 19:09 /home/licence.../TP
```

Remarque : Pour un répertoire, *r* signifie qu'on peut en voir le contenu (sans plus d'informations), *x* signifie qu'on peut s'y rendre (plus généralement, qu'on peut accéder à son contenu), et *w* signifie qu'on peut créer, supprimer ou renommer un élément du répertoire.

- Modification des droits d'accès de fichiers existants : *chmod*

chmod permet de modifier les droits d'accès d'un fichier, pourvu qu'on en

soit le propriétaire (ou qu'on soit le super-utilisateur).

Syntaxe :

chmod who op perm nomfichier

où * *who* est une combinaison des lettres *u*, *g* et *o*
(Rq : *a* est équivalent à *ugo*)

* *op* est un opérateur : + pour ajouter un droit, - pour supprimer un droit, et = pour réaliser une affectation absolue

* et *perm* est une combinaison des lettres *r*, *w* et *x*

Exemples :

```
$ chmod u+x /home/licence/durand/fic2

$ ls -l /home/licence/durand/fic2
-rwxrw-r--  1 mdurand     1150 Sep 26 18:59 /home/licence.../fic2

$ chmod go-r /home/licence/durand/fic2

$ ls -l /home/licence/durand/fic2
-rwx-w----  1 mdurand     1150 Sep 26 18:59 /home/licence.../fic2
```

Autre syntaxe possible :

chmod nombreoctal nomfic

où *nombreoctal* est un nombre en octal, formé en prenant en compte les codages suivants : x=1, w=2 et r=4

Exemples :

```
$ chmod 644 /home/licence/durand/fic2

$ ls -l /home/licence/durand/fic2
-rw-r--r-- 1 mdurand     1150 Sep 26 18:59 /home/licence.../fic2
```

- Droits d'accès à la création de futurs fichiers : *umask*

  Définition d'un masque de protection des fichiers (en octal) qui sera utilisé dans le futur, *lors de la création de nouveaux fichiers*.

  Cette commande indique les autorisations qu'on souhaite *enlever*. Attention à ne pas faire la confusion avec chmod !

Exemple : `$ umask 023`

les fichiers auront alors les permissions suivantes :

rwxr-xr--

- Changement de propriétaire : *chown*
  Changement de groupe : *chgrp*

  Permettent de modifier le propriétaire ou le groupe d'un fichier, pourvu qu'on en soit le propriétaire (ou qu'on soit le super-utilisateur).

  Exemples :

```
$ ls -l /home/licence/durand/fic2
-rwxrw-r--  1 mdurand     1150 Sep 26 18:59 /home/licence.../fic2

$ chown bdupond /home/licence/durand/fic2

$ ls -l /home/licence/durand/fic2
-rwxrw-r--  1 bdupond     1150 Sep 26 18:59 /home/licence.../fic2
```

III. QUELQUES COMMANDES UTILES

III.1 Quelques commandes générales

- *passwd* (ou yppasswd) : permet de changer son mot de passe

- *echo* : permet de faire afficher du texte à l'écran (noter qu'elle sera aussi utile pour voir les contenus des variables, voir plus loin)

  Exemple :
```
$ echo essai avec echo
essai avec echo
```

- *who* : permet de connaître les utilisateurs actuellement connectés

  Exemple :
```
$ who
     bdupond   ttyp2     Oct 9   09:21
     mdurand   ttyp4     Oct 9   10:17
```

- *man* : permet d'obtenir les informations relatives à une commande

  Exemple :
```
$ man ls
... affichage du manuel de "ls" page par page ...
$ man 2 write
... affichage du manuel de la fonction write ...
```

Note. Le manuel en ligne est organisé *en* ***sections***, dont on peut spécifier le numéro dans la commande :

1. commandes utilisateur
2. appels système (fournis par le noyau)
3. fonctions de bibliothèques (ex. fonctions sur chaînes)
4. fichiers spéciaux (notamment de /dev)
5. formats de fichiers et conventions (ex. /etc/passwd)
6. jeux
7. divers
8. administration du système

- **Exercice** : En supposant que vous êtes l'utilisateur bdupont et que vous venez de vous connecter sur un serveur, avec le répertoire de login figurant dans l'arborescence telle que donnée en exemple plus haut (p.4), écrire la suite de commandes qui vous permet successivement de :
  - vous déplacer dans le répertoire des masters (en chemin relatif, en chemin absolu)
  - voir un listing long du contenu de ce répertoire - Si l'utilisateur dont le répertoire de login est martin a tous les droits sur ce répertoire mais a enlevé tous les droits pour les autres utilisateurs, que verrez-vous au début de la ligne correspondante dans le listing long ?
  - en restant dans ce même répertoire, voir la liste des fichiers de source C se trouvant dans le répertoire de login de l'utilisateur mdurand (on suppose qu'il y en a, et que vous avez le droit de le faire)
  - retourner dans le sous-répertoire TP de votre répertoire de login
  - visualiser le chemin absolu de votre répertoire courant
  - visualiser le contenu du fichier fic1
  - enlever les droits d'écriture à tous les autres utilisateurs, pour tous les fichiers existants dont le nom commence par f
  - changer votre umask afin que tous les nouveaux fichiers que vous

créerez aient tous les droits pour vous et les utilisateurs de votre groupe, mais aucun droit pour les autres utilisateurs
  - créer un répertoire Backup, puis y déplacer tous les fichiers contenus dans /tmp (on suppose qu'il y en a, et en général tout le monde y a accès !)
  - voir la liste des utilisateurs actuellement connectés sur ce serveur
  - visualiser le manuel en ligne de la commande qui vous a permis de faire cela.

III.2 Commandes de manipulation de fichiers

- *wc* : permet de compter le nombre de lignes, de mots et de caractères dans un fichier

Exemple :
```
$ wc fic3
     5    32   167
```

- *head* : permet de faire afficher les *k* premières lignes d'un fichier

Exemple :
```
$ head -15 fic2
... affichage des 15 premières lignes de fic2 ...
```

- *tail* : permet de faire afficher soit les *k* dernières lignes d'un fichier (dans le cas d'un -), soit les lignes de la ligne n°*k* à la fin (dans le cas d'un -n +)

Exemples :
```
$ tail -6 fic3
... affichage des 6 dernières lignes de fic3 ...

$ tail -n +8 fic3
... affichage de fic3 de la ligne 8 à la fin ...
```

- *grep* : fait afficher les lignes d'un fichier qui contiennent une chaîne de caractères donnée (en fait, une expression régulière réduite)

Exemple :
```
$ grep salut fic2
... affichage de toutes les lignes de fic2 qui
contiennent "salut"

$ grep 'le langage C' Introduction
... affichage de toutes les lignes de
Introduction qui contiennent "le langage C" ...
```

- *find* : permet de trouver, dans *toute la sous-arborescence* du répertoire indiqué en paramètre, tous les fichiers satisfaisant un certain critère (spécifié via une ou des options). De plus l'option -exec permet de faire exécuter une commande sur chaque fichier ainsi trouvé.

  Syntaxe usuelle :

  find chemin expression

  où le *chemin* est l'emplacement à partir duquel la recherche se fait, et *l'expression* permet de spécifier le(s) critère(s) de recherche, dont :

-name nom	recherche par nom
-type typefic	recherche par type (d : répertoire, f : fichier ordinaire, …)
-user nom	recherche par propriétaire
-size taille	recherche par taille (en blocs de 512 octets)
-exec comm	provoque l'exécution de la commande *comm* sur le(s) fichier(s) trouvé(s)

  Exemples :

```
$ find . -name fic2
... recherche, dans la sous-arborescence du
répertoire courant, tous les fichiers dont le nom
est fic2 ...

$ find . -name "fic*"
... recherche, dans la sous-arborescence du
répertoire courant, tous les fichiers dont le nom
commence par "fic"

$ find /home/licence -user bdupond
... recherche, dans la sous-arborescence de
/home/licence, tous les fichiers dont le
propriétaire est bdupond ...

$ find Rapports -name "*.pdf" -exec lpr {} \;
... recherche, dans la sous-arborescence du
répertoire Rapports, tous les fichiers dont le
nom finit par .pdf et imprime ces fichiers ...
```


- **Exercice** : Ecrire la commande qui permet de rechercher tous les fichiers de la sous-arborescence du répertoire `Experimentations` dont le suffixe est `.log` et d'afficher à l'écran le contenu de chacun de ces fichiers.

  (aide : penser à l'option -exec de find)

IV. LES INTERPRETEURS DE COMMANDES

IV.1 Généralités

- L'interpréteur de commandes (**shell**) choisi par l'utilisateur (ou pour lui par l'administrateur) est activé lors de sa connexion.

  C'est un processus qui s'exécute, jusqu'à ce que l'utilisateur se déconnecte, et interprète les commandes pour que le système puisse les exécuter. En gros il exécute donc la boucle suivante :

  Faire
  saisir et interpréter commande
  exécuter commande
  jusqu'à commande de fin

- Des fichiers sont lus (et exécutés) à la connexion, ou à la création de nouveaux shells - ils sont extrêmement utiles pour configurer son environnement de travail :

  - en sh ou bash : à la connexion, exécution des fichiers */etc/profile* et *.profile* de l'utilisateur. En outre, au lancement d'un nouveau bash, il y a exécution du fichier *.bashrc*

  - en csh : à la connexion, exécution des fichiers *.cshrc* et *.login* de l'utilisateur. En outre, au lancement d'un nouveau csh, il y a exécution du fichier *.cshrc*

- Remarque : Lors de l'exécution d'une commande (externe au shell), un *processus fils* est créé pour son exécution. Nous détaillerons cela plus tard.
  Il y a aussi des commandes "internes" au shell, elles ne provoquent pas la création d'un nouveau processus (ce sont les "built-in" : cd, pwd, echo,…).

IV.2 Les variables

- Variables internes et variables d'environnement :
  - variables internes : elles ne sont connues que par le shell en cours d'exécution,
  - variables d'environnement : elles sont connues par le shell et par ses processus fils.

  Par convention, les variables internes sont en minuscules et les variables d'environnement sont *en majuscules*.

- Syntaxe des différentes opérations sur variables :

	en (ba)sh	en csh
1. affectation d'une variable interne :	`mavariable=bonjour`	`set mavariable=bonjour`
	`TERM=vt100`	`set TERM=vt100`
2. affectation d'une variable d'environnement :	`export TERM=vt100`	`setenv TERM vt100`
3. voir la valeur d'une variable :	`echo $mavariable`	`echo $mavariable`
4. obtenir la liste des variables et de leurs valeurs :	`set` (toutes les variables)	`set` (variables internes)
	`env` ou `printenv` (variables d'environnement)	`env` ou `printenv` (variables d'environnement)
5. supprimer des variables :	`unset mavariable`	`unset mavariable` (variables internes)
		`unsetenv TERM` (variables d'environnement)

- Principales variables : les "**variables d'environnement**" caractérisent

l'environnement d'exécution de votre shell, elles sont très importantes, notamment

- HOME : nom du répertoire de login (argument par défaut de *cd*)
- PATH : liste de noms de répertoires susceptibles de contenir des commandes (séparés par :)
- USER : nom d'utilisateur
- PWD : chemin d'accès du répertoire courant (automatiquement positionnée par *cd*)

IV.3 Redirections et tubes

• Généralités :
- entrée standard par défaut (stdin) : clavier
- sortie standard par défaut (stdout) : écran
- sortie erreur standard par défaut (stderr) : écran

• Syntaxe des redirections :

1. redirection de l'entrée standard sur fichier :

commande < fic

2. redirection de la sortie standard sur fichier :

commande > fic

3. redirection de la sortie en concaténation :

commande >> fic

4. redirection de la sortie erreur sur fichier :

commande 2> fic (stdout et stderr : commande >& fic)

5. tube entre commandes :

commande1 | commande2

Redirige la sortie standard de commande1 sur l'entrée standard de commande2 (ici il ne s'agit pas de redirection sur un fichier).

Remarque. Les tubes sont souvent utilisés avec les "filtres" : un "filtre"

est une commande qui lit ses données sur l'entrée standard et écrit ses résultats sur la sortie standard.

Exemple : `head -12 fic.txt | tail -2`

Donnons déjà une intuition de l'exécution de commande1 | commande2 (nous approfondirons plus tard) : les deux commandes sont *exécutées par des processus distincts*, qui utilisent un *tube* ("pipe") pour communiquer (le premier processus redirige sa sortie standard sur le tube, et le deuxième processus y redirige son entrée standard).

• Exemples : Que réalisent les commandes suivantes ?

```
cat fic1 fic2 fic3 > res
who | more
```

• **Exercice** : Les commandes suivantes ont-elles un sens ? Si oui, que signifient-elles précisément, et si non, pourquoi ?

```
ls -l | grep '^d' | more
tail +3 res.txt | who
file `tty`
```

IV.4 Substitution de commande

• La substitution de commande, par le backquote, permet l'utilisation du résultat d'une commande dans une affectation ou en paramètre d'une autre commande.

Exemples :
```
$ nbfic=`ls | wc -l`
$ echo `ls | wc -l`
```

IV.5 Gestion des tâches

• Pour lancer une tâche en background (et donc reprendre la main immédiatement), il suffit de terminer la commande par le caractère &

Exemple : `$ gcc monfic.c &`

Le shell affiche alors le numéro du processus créé et son numéro de job en background.
Note. Si le shell se termine, le processus en background se terminera aussi (en alternative -> utiliser nohup).

- Il est recommandé de rediriger la sortie standard et la sortie erreur standard (sur /dev/null ou sur un fichier) pour éviter les affichages intempestifs du processus en background.

- Le processus ne peut alors plus être interrompu par ^C, pour l'interrompre, il faudrait lui envoyer un signal à l'aide de la commande kill (voir plus loin).

IV.6 Processus et signaux

- La commande *ps* permet d'obtenir des renseignements sur l'état des processus en cours :
  - sans option : processus appartenant à l'utilisateur, et attachés à un terminal
  - option u : informations du type pourcentage mémoire, pourcentage CPU, etc…
  - option a : processus de tous les utilisateurs
  - option x : processus non attachés à un terminal

  Le PID (numéro de processus) est indiqué dans la première colonne.

  Exemple :

```
$ ps -au
USER       PID %CPU %MEM   SZ  RSS TT STAT    TIME COMMAND
bdupond   3548  4.5  3.5 6028  864 p1 S       0:08 vi fic.c
mdurand   3547  3.7  4.3 1864 1060 p1 S       0:08 gcc f1.c
ymartin   3560  0.0  3.0 5884  740 p2 S       0:07 emacs
laurence  3568  0.0  2.1  696  524 p3 R       0:00 ps -au
bdupond   3527  0.0  0.7  368  168 p0 I       0:00 -csh (csh)
```



```
laurence  3551  0.0  2.5  680  620 p3 S       0:00 -zsh (zsh)
ymartin   3546  0.0  0.2   48   44 p1 I       0:00 -csh (csh)
mdurand   3518  0.0  0.7  368  168 p0 I       0:00 -csh (csh)
```

- Un **signal** est une sorte de notification envoyée à un processus (permettant de lui signaler un événement), qui fournit donc une forme limitée de communication entre processus.
  (nous y reviendrons plus tard)

- La commande *kill* permet d'envoyer un signal à un processus. En particulier :
  - le signal n°2 (SIGINT) correspond à ^C
  - le signal n°3 (SIGQUIT) correspond à ^\
  - le signal n°9 (SIGKILL) provoque la **mort** de façon certaine.

  Syntaxe :

  kill -n°signal pid

  Exemple : `$ kill -9 12856`

- La commande *trap* permet d'intercepter des signaux, sauf les signaux n°9 (SIGKILL) et SIGSTOP (généralement 17) :

  trap com sig

  à l'interception du signal *sig*, exécution de la commande *com* (si aucune commande n'est indiquée, i.e. argument "", alors le shell ignore le signal).

  Exemple : `trap "echo reception ctrl C" 2`

PARTIE 2. PROGRAMMATION SYSTEME

V. RAPPELS POUR LA PROGRAMMATION C

V.1 Arguments de la fonction main

- Une fois compilé, un programme C donne naissance à une commande, que vous activez en tapant le nom correspondant.

Exemples :
```
$ gcc monprog.c
$ ./a.out
$ gcc -o monexec monprog.c
$ ./monexec
```

Note. Pourquoi utilisons-nous ./ ?

- De plus, si l'on souhaite passer des paramètres (ou même des options) sur la ligne de commande, on va avoir recours aux **arguments de la fonction *main***.

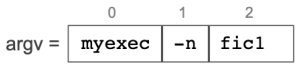
  En effet, l'en-tête de *main* peut être le suivant :

*int main(int argc, char *argv[]);*

où

- *argc* correspond au nombre d'arguments +1 (c'est à dire le nombre de mots sur la ligne de commande, nom de commande compris),
- *argv* est un tableau de chaînes de caractères, la première est le nom de la commande, et les autres sont les arguments.

Exemple :
```
$ myexec -n fic1
```
alors `argc=3, argv[0] = "myexec", argv[1] = "-n", et argv[2] = "fic1"`



- Exemple : Affichage de la liste des arguments

```
int main(int argc, char **argv){
  int i;
  for(i=0; i<argc; i++)
     puts(argv[i]);
  return 0;
}
```

ou

```
int main(int argc, char **argv){
   while (*argv)
      puts(*argv++);
   return 0;
}
```

V.2 Les E/S de haut niveau sur fichiers

- De manière générale, les fichiers sont considérés comme des "flots" qui peuvent se présenter :
  - soit sous la forme de *fichiers de texte,*
  - soit sous la forme de *fichiers binaires.*

  Les fonctions d'entrée/sortie de haut niveau sont employées lorsque le fichier est traité comme un fichier de texte, dans lequel les données sont considérées comme structurées.

- Les flots sont alors représentés par des variables de **type FILE ***, on aura des déclarations de la forme :

*FILE *fic;*

Trois flots sont prédéfinis dans <stdio.h> (et automatiquement ouverts) :
- *FILE *stdin* : entrée standard,
- *FILE *stdout* : sortie standard,
- *FILE *stderr* : sortie erreur standard.

- Ouverture de fichier - La fonction *fopen* :

*FILE *fopen(const char *nomfic, const char *mode);*

où

- *nomfic* est le nom physique du fichier,
- *mode* est le mode d'ouverture du fichier, notamment : "r" pour lecture, "w" pour écriture, "a" pour écriture à la fin du fichier.

fopen ouvre le fichier selon le mode indiqué et rend en résultat un pointeur sur le flot correspondant (ou NULL si échec).

- Les fonctions *fprintf* et *fscanf* :

  Elles se comportent comme *printf* et *scanf*, mais prennent comme premier paramètre une variable de type FILE * qui indique le flot sur lequel écrire ou lire.

  Exemples :
```
fprintf(stderr, "Erreur d'ouverture : %s\n", nomfic);
fscanf(pfic,"%d %c", &x, &c);
```

- Les fonctions *fgets* et *fputs* :

  *char *fgets(char* s, int n, FILE *fic);*

  lecture d'une chaîne de caractères sur le flot *fic* et stockage dans *s*, arrêt lorsque *n-1* caractères sont lus ou fin de ligne. Le caractère '\0' est placé à la fin de la chaîne.
  Elle rend en résultat la chaîne *s*, ou NULL si erreur.

  int fputs(const char s, FILE *fic);*

  écriture de la chaîne de caractères *s* sur le flot *fic*.
  Elle rend en résultat une valeur non négative, ou EOF si erreur.

- Les fonctions *fgetc*, *fputc*, et *ungetc* :

  *int fgetc(FILE *fic);*

  lecture du caractère suivant sur le flot *fic*.
  Elle rend en résultat le caractère lu, ou EOF si erreur ou fin de fichier.

  *int fputc(int c, FILE *fic);*

écriture du caractère *c* sur le flot *fic*.
Elle rend en résultat le caractère écrit, ou EOF si erreur.

*int ungetc(int c, FILE *fic);*

"délecture" du caractère *c* sur le flot *fic*.

- La fonction *fflush* :

  *int fflush(FILE *fic);*

  écriture immédiate du tampon courant. Elle rend 0, ou EOF si erreur.

- La fonction *fclose* :

  *int fclose(FILE *fic);*

  close ferme le fichier représenté par *fic*.

V.3 Les E/S de bas niveau sur fichiers

- Dans le système Unix, les fichiers sont gérés par les processus grâce à une *table de descripteurs de fichiers ouverts.*
  Lorsque l'on utilise les entrées/sorties de bas niveau (lectures/écritures de blocs d'octets), les fichiers sont représentés par un *indice dans cette table*, c'est-à-dire un entier. Dans la suite (prog. système), **nous aurons exclusivement recours à ces E/S présentées ci-dessous**, car les contenus des "fichiers" ne seront vus que comme des suites d'octets.

  L'entrée standard (*stdin*), la sortie standard (*stdout*) et la sortie erreur standard (*stderr*) sont associées respectivement aux indices 0, 1 et 2.

- Ouverture de fichier - La fonction *open* va permettre, lors de l'ouverture d'un fichier, de lui associer un nouvel indice dans la table :

  *int open(const char *nomfic, int mode, int perm);*

  où
  - *nomfic* est le nom physique du fichier,
  - *mode* est le mode d'ouverture

(O_RDONLY : ouverture en lecture, O_WRONLY : ouverture en écriture, O_RDWR : ouverture en lecture/écriture)

- *perm* est optionnel : en cas de création, il indique les droits donnés sur le nouveau fichier.

open ouvre le fichier selon le mode indiqué (ou le crée s'il n'existait pas et si le mode est écriture), et lui associe un indice dans la table des descripteurs, qui est rendu en résultat de la fonction (si l'ouverture est impossible, la fonction rend -1).

- La fonction *creat* :

int *creat*(char *nomfic, int perm);

où

- *nomfic* est le nom physique du fichier,
- *perm* indique les droits donnés sur le fichier.

creat crée le fichier *nomfic*, avec les permissions indiquées, et lui associe un indice dans la table des descripteurs, qui est rendu en résultat de la fonction (si la création est impossible, la fonction rend -1).

- La fonction *read* :

*int read(int ficno, void *tmp, size_t nbre);*

où

- *ficno* est le numéro de descripteur associé au fichier, tel qu'il a été donné par *open* ou *creat*,
- *tmp* est le tampon où seront rangés les octets lus,
- *nbre* est le nombre d'octets à lire.

read lit *nbre* octets dans le fichier représenté par *ficno*, et les range dans le tampon *tmp*. Elle renvoie en résultat le nombre d'octets effectivement lus, 0 sur fin de fichier ou -1 sur erreur.

- La fonction *write* :

*int write(int ficno, const void *tmp, size_t nbre);*

où

- *ficno* est le numéro de descripteur associé au fichier, tel qu'il a été donné par *open* ou *creat*,
- *tmp* est le tampon où se trouvent les octets à écrire dans le fichier,
- *nbre* est le nombre d'octets à écrire.

write écrit *nbre* octets, se trouvant dans le tampon *tmp*, dans le fichier représenté par *ficno*. Elle renvoie en résultat le nombre d'octets effectivement écrits, ou -1 sur erreur.

- La fonction *close* :

int close(int ficno);

close ferme le fichier représenté par *ficno*.

- Exemple élémentaire : Lecture d'un seul tampon de 512 octets dans un fichier et écriture dans un autre.

```
int main(int argc, char **argv){
 int source, dest;
 int n;
 char tampon[512];
 if (argc==3){
    if (((source=open(argv[1], O_RDONLY)) != -1) &&
        ((dest=open(argv[2], O_WRONLY|O_CREAT, 0644)) != -1)){
         n=read(source, tampon, 512);
         write(dest, tampon, n);  //  /!\ utilisation de n
         close(dest);
         close(source);
         return 0;
    }
    else {
       fprintf(stderr, "Pb d'ouverture de fichiers\n");
       return -1;
    }
 }
 else {
    fprintf(stderr, "Mauvais nbre d'arguments\n");
    return -1;
 }
}
```

NB. en règle générale, la recopie de l'intégralité d'un fichier se réalise en répétant de telles lectures/écritures, bloc par bloc, dans une boucle.

V.4 Gestion des messages d'erreur : errno et perror

- Lorsqu'une erreur se produit lors de l'exécution d'une fonction système ou d'une fonction de bibliothèque C (par exemple *open*), la variable *errno* est automatiquement positionnée, elle va permettre de connaître lisiblement le type d'erreur qui s'est produit.
  (vous pouvez vérifier si le man en ligne de la primitive indique bien que *errno* est positionné lors d'une erreur)

- La variable globale *sys_errlist* est un tableau de chaînes de caractères, ce tableau contient tous les messages d'erreur possibles.
  La valeur de *errno* donne l'indice dans cette table du message correspondant à l'erreur qui vient de se produire.

- Le message peut donc être affiché en récupérant *sys_errlist[errno]*
  Il n'est pas nécessaire de faire ce traitement soi-même, la fonction

  *void perror(const char *s);*

  a été prédéfinie dans ce but.

  Cette fonction affiche le message d'erreur, précédé de la chaîne de caractères *s* si celle-ci ne vaut pas NULL.

Exemple :

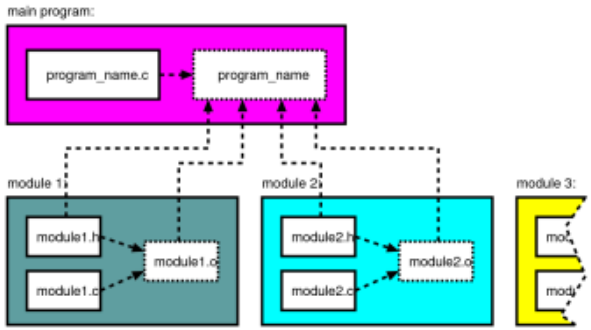
```
if ((source=open(argv[1], O_RDONLY)) != -1)
  ...
  else perror("open");
```

Attention, l'utilisation de cette fonction n'a bien sûr pas de sens si l'erreur n'est pas produite par une primitive capable de positionner *errno*. Par exemple, pour gérer une erreur lors de l'accès à un tableau, on utilisera classiquement un *fprintf* sur *stderr*.

VI. SUPPLEMENT - PROGRAMMATION MODULAIRE

VI.1 Organisation modulaire - compilation

- Dès qu'une application atteint un certain niveau de complexité et une certaine taille, il est bon de prévoir une organisation modulaire, c'est à dire que l'application n'est pas entièrement placée dans un unique fichier source C.
  L'organisation se présente alors sur le modèle qui suit, avec fichiers implémentations et headers :



https://www.icosaedro.it/c-modules.html

La fonction *main* est placée dans le fichier apparaissant ici sous le nom program_name.c, elle est aussi succincte que possible et fait appel à d'autres fonctions définies dans les autres fichiers .c

Chaque fichier d'entête (ou "header", fichier .h) contient les déclarations des fonctions qui sont définies dans son fichier implémentation, et toute autre déclaration nécessaire à ces fonctions (constantes, types utilisateur,…).

En outre, il est bon que le header contienne la *documentation* relative à ces fonctions (informations sur l'usage de ces fonctions - on pourra adopter un format de commentaires permettant de générer automatiquement et facilement une documentation HTML par exemple, avec

Doxygen, http://www.doxygen.org).

- Exemple : nous utiliserons l'exemple simple suivant (mais attention, les fichiers d'entête ne sont pour l'instant pas vraiment correctement écrits)

Fichier convertir.h :

```
// taux de conversion
#define TAUX 1.28
// type de conversion
#define EURO 0
#define DOLLAR 1

void dollar_euro(double dollar);
void euro_dollar(double euro) ;
```

Fichier convertir.c :

```
#include <stdio.h>
#include "convertir.h"

void dollar_euro(double dollar) {
  printf("\t%.4lf dollars = %.4lf euros\n", dollar,
  dollar / TAUX);
}

void euro_dollar(double euro) {
  printf("\t%.4lf euros = %.4lf dollars\n", euro,
  euro * TAUX);
}
```

Fichier outils.h :

```
#define CONTINUER 1
#define SORTIR 0

void Message(char *message, short choix);
void LireDouble(char *phrase, double *nombre);
void LireShort(char *phrase, short *nombre);
```

Fichier outils.c :

```
#include <stdio.h>
#include <stdlib.h>
#include "outils.h"

void Message(char *message, short choix) {
  printf("%s", message);
  if(choix == SORTIR) {
    printf("\n ... interruption du programme \n");
    exit(1);
  }
}
```



```
void LireDouble(char *phrase, double *nombre) {
  Message(phrase, CONTINUER);
  if (scanf("%lf", nombre) == 0)
    Message("Vous n'avez pas entre un float", SORTIR);
}

void LireShort(char *phrase, short *nombre) {
  Message(phrase, CONTINUER);
  if (scanf("%hd", nombre) == 0)
    Message("Vous n'avez pas entre un entier", SORTIR);
}
```

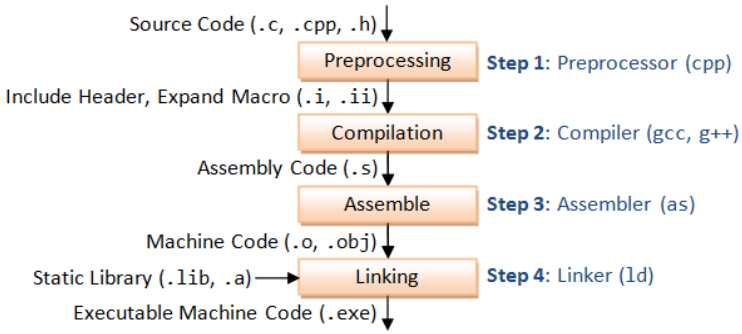
Fichier monpg.c :

```
#include <stdio.h>
#include "convertir.h"
#include "outils.h"

int main (int argc, char **argv) {
  double val;
  short action, type;
  action = CONTINUER; // initialisation
  do {
    LireDouble("entrez un nombre : ",  &val);
    if (val == 0)
      action = SORTIR;
    else {
      LireShort("convertir en euros (0) / dollars (1) ? ",
                &type);
      switch (type) {
      case EURO  : dollar_euro(val); break;
      case DOLLAR : euro_dollar(val); break;
      default : Message("Euros (0) /Dollars (1) ?",
                        CONTINUER);
      }
    }
  }
  while (action == CONTINUER);
  return 0;
}
```

• Rappelons le **processus de compilation** : Lorsque vous compilez une application C (avec gcc), *4 opérations* sont en fait effectuées successivement :

- passage au préprocesseur (*preprocessing*)
- compilation en assembleur, puis conversion en code machine
- édition de liens



https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html

NB. Pendant la phase d'édition de liens, les *fichiers objects* (produits par les phases précédentes) sont combinés pour créer le *code exécutable*.
A cette étape, il peut être nécessaire d'avoir recours également à des *bibliothèques*. Elles contiennent des définitions de fonctions, elles sont soit "statiques" soit "dynamiques" (partagées).
Les identifiants de bibliothèques statiques sont de la forme libXXXX.a (où XXXX est le nom de la bibliothèque), et ceux des bibliothèques dynamiques sont de la forme libXXXX.so. Pour utiliser la bibliothèque à l'édition de liens, se servir de l'option `-lxxxx` de gcc
L'option `-L` de gcc permet en outre de spécifier des chemins de recherche des bibliothèques.

- On pourra procéder à la **compilation séparée** de l'application donnée en exemple ci-dessus par la suite de commandes suivante (nous verrons plus élaboré plus loin) :

```
gcc -c convertir.c
gcc -c outils.c
gcc -c monpg.c
gcc convertir.o outils.o monpg.o -o monpg
```

ce qui donnera alors naissance au fichier exécutable `monpg`.

L'option **`-c`** de gcc arrête le processus de compilation à la génération du

fichier objet .o (**pas** d'édition de liens). Il reste à procéder à l'édition de liens entre tous les fichiers objets, pour produire l'exécutable (dernière commande ci-dessus).

VI.2 Le préprocesseur : compilation conditionnelle

- Des **directives à l'attention du préprocesseur** permettent d'inclure ou pas des portions du fichier source dans le texte qui sera produit pour le compilateur.
  Les conditions mises en jeu peuvent porter sur :
  - l'existence ou pas de symboles,
  - la valeur d'une expression.

- Directives relatives à l'existence de symboles :

```
#ifdef symbole
        suite-instructions1
#else
        suite-instructions2
#endif
```

Si le symbole considéré est défini (i.e. a fait l'objet d'un *#define*), alors suite-instructions$_1$ est incorporé au texte à compiler, sinon suite-instructions$_2$ est incorporé.

Remarque : pour tester la non-définition d'un symbole, on utilise la directive *#ifndef*

- Directives relatives à la valeur d'une expression :

```
#if condition                #if condition1
…                            …
#else                        #elif condition2
…                            …
#endif                       #endif
```

Suivant la valeur de la (ou des) condition(s), on choisit d'incorporer un ensemble d'instructions ou un autre. Les conditions s'expriment au

moyen des opérateurs relationnels, arithmétiques et logiques de C.

- Exemple : Ces constructions de compilation conditionnelle sont très utiles (même indispensables !) dans la définition des fichiers d'entête (headers), pour éviter les inclusions multiples.
  Attention : on rappelle que, dans l'exemple utilisé en début de section, les fichiers d'entête ne sont pas conçus totalement correctement. On devrait avoir par exemple pour le fichier convertir.h :

```
#ifndef CONVERTIR_H
#define CONVERTIR_H

// taux de conversion
#define TAUX 1.28
// type de conversion
#define EURO 0
#define DOLLAR 1

void dollar_euro(double dollar);
void euro_dollar(double euro) ;

#endif
```

VI.3 Gestion des dépendances : make

- Nous venons de voir que deux fichiers essentiels peuvent découler de la compilation d'un fichier source fic.c: le *fichier objet* fic.o, et le *fichier exécutable* généré en fin de compilation (après édition de liens).

  Lorsque le fichier fic.c est modifié, ces deux fichiers deviennent obsolètes, il faut les recréer (de même si fic.c inclut un fichier fic.h et que celui-ci est modifié).

- Lorsqu'un programme se compose de plusieurs fichiers sources, il faut pouvoir gérer les dépendances entre ces fichiers de façon à procéder à des re-compilations lorsque nécessaire.

  C'est l'utilitaire *make* qui permet de gérer ces dépendances, elles sont décrites dans un fichier généralement appelé *makefile* (ou *Makefile*).

- Les *règles de dépendance* d'un fichier *makefile* sont généralement de la forme :

  but_1 but_2 ... but_i : $nom\text{-}fic_1$ $nom\text{-}fic_2$ ... $nom\text{-}fic_j$
  $commande_1$
  ...
  $commande_k$

  ce qui signifie que les fichiers "buts" but_1 but_2 … but_i sont dépendants des fichiers "antécédents" $nom\text{-}fic_1$ $nom\text{-}fic_2$ … $nom\text{-}fic_j$.

  Une telle règle de dépendance est **interprétée de la façon suivante** : mettre les fichiers $nom\text{-}fic_1$ $nom\text{-}fic_2$ … $nom\text{-}fic_j$ à jour si nécessaire (en prenant en compte leurs propres règles de dépendance), puis exécuter les commandes $commande_1$ … $commande_k$

  A noter toutefois que la cible (partie à gauche du :) et les antécédents ne sont pas nécessairement des noms de fichiers, ils peuvent être de simples étiquettes. Il est souhaitable d'associer les cibles qui ne sont pas des noms de fichiers à une règle spéciale, de cible .PHONY (sans commandes).

- Exemple : pour l'exemple de l'application de conversion vu auparavant, on pourrait avoir simplement

```
all: monpg

monpg: convertir.o outils.o monpg.o
	gcc convertir.o outils.o monpg.o -o monpg

monpg.o : monpg.c
	gcc -c monpg.c

convertir.o : convertir.c
	gcc -c convertir.c

outils.o : outils.c
	gcc -c outils.c

clean :
	@echo "On efface les fichiers objets"
	-rm *.o
```

- Si la commande *make* est utilisée avec un paramètre, il doit correspondre à un membre gauche d'une règle de dépendance, et c'est cette règle qui est exécutée.
  Si la commande est utilisée sans paramètre, c'est la *première règle* de dépendance qui est exécutée.
  En outre, l'option `-f fic` permet d'indiquer qu'on souhaite utiliser le fichier `fic` au lieu du fichier `makefile`, et l'option `-n` permet d'afficher les commandes qui devraient s'exécuter, sans les exécuter.

Exemple :

```
$ make outils.o
gcc -c outils.c
$ make
gcc -c convertir.c
gcc -c monpg.c
gcc convertir.o outils.o monpg.o -o monpg
$ make clean
On efface les fichiers objets
rm *.o
$ make
gcc -c convertir.c
gcc -c outils.c
gcc -c monpg.c
gcc convertir.o outils.o monpg.o -o monpg
```

- Un fichier *makefile* peut aussi contenir des définitions de macros, en utilisant la syntaxe :

  nom_macro=valeur_de_la_macro

  Elles sont alors utilisées sous la forme *$(nom_macro)* ou *${nom_macro}*

  Il existe aussi des *macros spéciales*, surtout utiles dans le cadre des règles implicites (voir ci-dessous), notamment :

  $@ nom de la cible
  $? liste des noms des dépendances changées (i.e. fichiers plus récents que la cible)
  $< nom du fichier qui est la première dépendance

- On remarque que toutes les règles permettant d'obtenir un fichier objet à partir de son fichier source sont évidemment bâties sur le même modèle. Pour nous éviter de les écrire, il existe des "règles de suffixes" implicites, en particulier celle qui *associe un .o avec son .c* :

```
.c.o:
     $(CC) $(CFLAGS) $(CPPFLAGS) -c $<
```

  `CC` étant le compilateur C, `CFLAGS` des options pour le compilateur, et `CPPFLAGS` des options pour le préprocesseur. Il suffit donc de les définir (si nécessaire) selon le besoin.

- On pourra définir d'autres macros, comme celle contenant la liste de tous les noms de fichiers objets, OBJS.

Exemple :

```
CC = gcc
OBJS = convertir.o outils.o monpg.o
EXEC = monpg

all: $(EXEC)

$(EXEC): $(OBJS)
        $(CC) $(OBJS) -o $(EXEC)

clean :
        @echo "On efface les fichiers objets"
        rm $(OBJS)
```

Ce qui donne à l'utilisation :

```
$ make
gcc    -c -o convertir.o convertir.c
gcc    -c -o outils.o outils.c
gcc    -c -o monpg.o monpg.c
gcc convertir.o outils.o monpg.o -o monpg
```

- Voyons un dernier exemple, qui utilise la *substitution de suffixes*, et d'autre part suppose que les *headers sont dans un sous-répertoire* `include` :

```
CC = gcc
CPPFLAGS = -Iinclude
SRCS = convertir.c outils.c monpg.c
```

```
OBJS = $(SRCS:.c=.o)
EXEC = monpg

all: $(EXEC)

$(EXEC): $(OBJS)
        $(CC) $(OBJS) -o $(EXEC)

clean :
        @echo "On efface les fichiers objets"
        rm $(OBJS)
```

Ce qui donne à l'utilisation :

```
$ make
gcc -Iinclude   -c -o convertir.o convertir.c
gcc -Iinclude   -c -o outils.o outils.c
gcc -Iinclude   -c -o monpg.o monpg.c
gcc convertir.o outils.o monpg.o -o monpg
```

VII. PROGRAMMATION SYSTEME EN C

VII.1 Quelques mots sur les processus

- Un processus comprend essentiellement 3 zones en mémoire : la zone instructions, la zone données, la zone de la pile d'exécution.
  D'autre part, un certain nombre d'informations sont associées à ce processus : numéro d'utilisateur, descripteurs des fichiers ouverts, comportement vis à vis des signaux, …

- La seule façon de créer de **nouveaux processus** est de faire créer des *processus "fils"* par les processus existants.
  Ceci se fait au moyen de la fonction ***fork*** qui a pour effet de créer un processus fils en tous points identique à son père (sauf son pid).

- Le processus fils aura ensuite la possibilité d'exécuter un nouveau programme. Ceci se fait grâce à une fonction de la famille des *exec*

  Dans ce cas : ses zones instructions, données et pile se trouvent modifiées, mais l'environnement (numéro d'utilisateur, descripteurs des fichiers ouverts, comportement vis à vis des signaux, …) est inchangé.

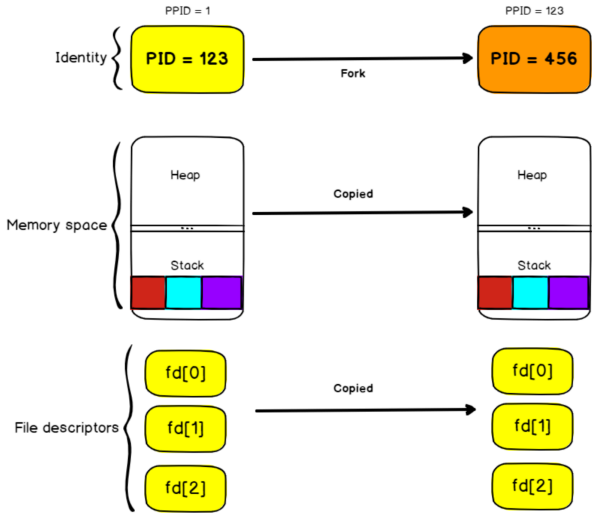
VII.2 La fonction *fork*

- La fonction *fork*

  int fork();

  permet à un processus de créer un processus fils qui est une *exacte copie* de lui-même (un clone), ayant bien sûr un identifiant (PID) différent.
  En particulier, ce fils *exécute le même code* que son père.

  D'autre part, ce fils a sa propre copie des descripteurs de fichiers de son père, il pourra donc les gérer indépendamment (il pourra ouvrir de nouveaux fichiers, ou fermer des fichiers ouverts sans que cela affecte son père).



https://devconnected.com/understanding-processes-on-linux/

Attention toutefois, *le résultat rendu par fork dans le processus fils et dans le processus* père va permettre de les différencier et de leur faire exécuter des tâches différentes.

- Valeurs de retour de la fonction *fork* :

  En cas de succès, *fork* retourne :

  - 0 dans le processus fils,
  - le PID du processus fils dans le processus père.

  et en cas d'échec, elle retourne -1 (et *errno* est positionné).

- Emploi habituel de cette valeur de retour :

```
switch (fork()){
   case -1 :
      fprintf(stderr, "Erreur de fork\n"); exit(-1);
   case 0 :
      /* comportement du fils */
      ...
   default :
      /* comportement du père */
      ...
}
```

VII.3 Les fonctions de la famille des *exec*

- Les fonctions *execl*, *execv*, *execle*, *execlp* et *execvp* sont de la même famille. Elles permettent de faire exécuter un nouveau programme à un processus (généralement après un *fork*) :

  *int execv(char *path, char *argv[]);*

  *int execlp(char *file, char *arg$_0$, char *arg$_1$, ... , char *arg$_n$, (char *)0);*

  *int execvp(char *file, char *argv[]);*

  où :

  - *path* est le nom absolu de la commande à exécuter,
  - *file* est un nom de fichier correspondant à la commande à exécuter, et cette commande sera cherchée en utilisant la variable PATH,
  - *argv* est un tableau qui contient la ligne de commande à exécuter (commande + options et paramètres), ou bien *arg$_0$, arg$_1$, ..., arg$_n$* correspondent aux chaînes de caractères sur cette ligne.

- Valeur de retour :

  Si la commande réussit, on ne revient pas d'une telle commande. Si la commande échoue, la valeur de retour est -1 (et *errno* est positionné).

- Exemples :

```
char *macommande[4] = { "ls", "-al", "../TP", (char *)0 };
...
execvp("ls", macommande);
```

  ou

```
execlp("ls", "ls", "-al", "../TP", (char *)0);
```

- Utilisation en conjonction avec le *fork* :

  Lorsqu'un processus a créé un processus fils, on peut être amené à faire exécuter une nouvelle commande à ce fils, de la façon suivante

```
switch (fork()){
   case -1 :
      fprintf(stderr, "Erreur de fork\n");
      exit(-1);
   case 0 :
      /* comportement du fils */
      ...
      if (execvp(nomcom, com) == -1) {
          perror("execvp");
          exit(-1);
      }
   default :
      /* comportement du père */
      ...
}
```

VII.4 La fonction *wait*

- Un processus père doit attendre (ou entériner) la "mort" de ses fils avant de "mourir" à son tour, cela grâce à la fonction *wait* (qui va permettre au système de libérer les ressources associées aux fils).
  Attention, tout fils pour lequel un *wait* n'a pas été réalisé va rester dans le système comme "zombie" (à la mort du père, il sera "adopté" par le processus *init*, qui fera le *wait* nécessaire).

- Lorsqu'un fils "meurt", le père reçoit un signal SIGCHLD. La fonction *wait*

*int wait(int *status);*

permet au processus père de *rester bloqué en attente de la réception d'un signal SIGCHLD*.

- Valeur de retour :

En cas de succès, *wait* retourne le numéro de PID du processus fils qui vient de "mourir", et s'il n'y a plus de processus fils, elle retourne -1.

Pour *attendre la mort de tous les fils*, on pourra donc faire une boucle de la forme :

```
for(i=1; i<=nb_fils; i++) wait(NULL);
```

ou même

```
while (wait(NULL)!=-1);
```

- Paramètre de *wait* :

Il est possible de passer en paramètre de *wait* l'adresse d'un entier dans lequel sera stocké l'état de retour du processus fils qui vient de "mourir".

- Améliorons alors le programme proposé :

Le programme précédent peut être amélioré, en faisant attendre la "mort" du processus fils

```
switch (fork()){
    case -1 :
       fprintf(stderr, "Erreur de fork\n");
       exit(-1);
    case 0 :
       /* comportement du fils */
       ...
       if (execvp(nomcom, com) == -1) {
           perror("execvp");
           exit(-1);
       }
    default :
       /* comportement du père */
       ...
       printf("Le fils %d est mort\n", wait(NULL));
 }
```


VII.5 Etat de retour d'un processus – Fonction *exit*

- La fonction *exit*

void exit(int status);

permet à un processus de "se suicider" en renvoyant un code de retour qui indique "dans quelles conditions il est mort" (0 indique généralement que tout s'est bien passé et que le processus est "mort" normalement à l'issue du traitement qu'il devait faire).
C'est ce code qui peut être récupéré par le paramètre de *wait*

- Ce code peut être interprété comme un entier codé sur 2 octets :

  - si l'octet bas (les 8 bits de poids faible) est à 0, alors le processus est "mort" par un appel à *exit*, et l'octet haut (les 8 bits de poids fort) indique la valeur qui a été passée en argument de *exit*

  - si l'octet bas est différent de 0, alors il y a eu réception d'un signal dont le numéro est indiqué par les 7 bits de poids faible de l'octet bas.

    De plus, si le bit de poids le plus fort de l'octet bas est à 1, alors il y a eu création d'un fichier "core".

- Améliorons encore notre programme :

```
switch (fork()){
    case -1 :
       fprintf(stderr, "Erreur de fork\n");
       exit(-1);
    case 0 :
       /* comportement du fils */
       ...
       if (execvp(nomcom, com) == -1){
           perror("execvp");
           exit(-1) ;
        }
    default :
       /* comportement du père */
       ...
       num = wait(&status) ;
       printf("Le fils %d est mort\n", num);
```

```
        // analyse du code de retour récupéré dans status:
        traitement_status(status) ;
        ...
}
```

VII.6 Interception des signaux – Fonction *signal*

- Un "signal" est produit par l'arrivée d'un événement anormal durant l'exécution d'un programme (erreur lors d'un accès mémoire, division par 0, …), ou peut être envoyé par un autre processus ou par l'utilisateur.

  La liste des signaux est donnée dans <signal.h>

- Par défaut, dans la plupart des cas, la réception d'un signal provoque la mort du processus en cours.

  La fonction *signal* permet de définir un comportement à adopter en cas de réception d'un signal donné (sauf pour les signaux SIGKILL et SIGSTOP) : déroutement sur une fonction "handler", ou ignorer, ou revenir au comportement par défaut.

  Elle prend en paramètres un numéro de signal, et un pointeur sur une fonction à exécuter en cas de réception du signal (cette fonction prend en paramètre un entier, qui sera le numéro de signal).

  A noter que :

  - si cette fonction est SIG_IGN : le signal est seulement ignoré,
  - si elle est SIG_DFL : le comportement par défaut est réinstauré.

- Exemple :

```
void intercept(int sig){    // handler pour SIGQUIT
   fprintf(stderr, "Reception signal %d\n", sig);
    ...
}

int main(){
  ...
  signal(SIGINT, SIG_IGN);
  signal(SIGQUIT, intercept);
  ...
}
```


- A noter que l'envoi d'un signal à un processus (du même utilisateur) peut se faire par la fonction :

  int kill(int pid, int sig);

  La fonction *kill* permet d'envoyer le signal *sig* au processus de numéro *pid*. En cas de succès la valeur de retour est 0, et en cas d'échec la valeur de retour est -1 (et *errno* est positionné).

  Cette fonction pourra être utile par exemple pour transmettre un signal à un processus fils dans un "handler" (cas de la programmation d'un shell).

VII.7 Squelette d'un shell

- Nous pouvons maintenant présenter la structure d'un shell. Un shell (très simplifié) peut être construit sur le squelette suivant :

```
Lecture d'une commande;
Tant que la commande de sortie n'est pas tapée
    Analyse de la commande;
    Si la commande correspond à une fonction interne
    alors   application de cette fonction aux arguments de la
            commande;
    sinon   création d'un processus fils qui exécutera la commande;
            et le processus père attend la mort du processus fils;
    Finsi
    Lecture d'une commande;
Fintantque
```

- D'autres aspects pourront être pris en compte, notamment :
  - les redirections et tubes,
  - l'enchaînement de plusieurs commandes sur une même ligne (avec le ";"),
  - la possibilité d'exécuter la commande en "background" (i.e. comportement particulier vis à vis des signaux, restauration de

l'environnement de départ, …)

- Une implémentation simple possible :

```
void mon_shell(){
  int no, status;
  char cmd[100];
  char *com[10];

  signal(SIGINT,SIG_IGN);

  printf("Voici mon shell, taper Q pour sortir\n");
  printf("> ");
  gets(cmd);

  while (strcmp(cmd,"Q")!=0){
     // lecture/analyse de la commande, création tableau com:
     Lire_commande(cmd,com);
     // execution:
     /* Pour les commandes internes : */
     if (strcmp(com[0],"cd")==0){
        if (com[1]==NULL) chdir(getenv("HOME"));
        else
          if (chdir(com[1])==-1) perror(com[1]);
     }
     else
        if (strcmp(com[0],"setenv")==0)
           setenv2();  // notre version
        // etc...
        else
           /* Pour les autres : */
           switch (fork()){
              case -1 :
                 perror("fork"); exit(-1);
              case 0 :
                 signal(SIGINT,SIG_DFL);
                 // "During an exec, the dispositions of
                 // handled signals are reset to the default;
                 // the dispositions of ignored signals are
                 // left unchanged" (man de signal)
                 if (execvp(com[0],com) == -1){
                    perror("execvp");
                    exit(-1) ;
                 }
              default :
                 no = wait(&status);
                 printf("Etat de retour du processus %d : %d\n",
                         no, status);
                 traite_status(no,status);
           }
     printf("> ");
     gets(cmd);
  }
}
```



```
int main(int argc,char *argv[]){
   mon_shell();
   return 0;
}
```

VII.8 Numéro d'un processus et de son processus père

- Les fonctions *getpid*

  int getpid();

  et *getppid*

  int getppid();

  permettent à un processus de connaître son PID ainsi que le PID de son processus père.

  NB. La fonction *getpid* permettra notamment de générer des noms de fichiers temporaires uniques.

- Rappel : un processus connait le PID de chacun de ses processus fils, c'est l'entier ramené en résultat de la fonction *fork*

VII.9 Création et manipulation de tubes

- La fonction *pipe*

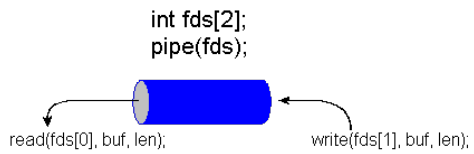
  int pipe(int fd[2]);

  permet de créer un *canal de communication (tube)* entre des processus.

  Ce "tube" (pipe) possède deux "extrémités" fd[0] et fd[1] qui sont des *descripteurs de fichiers*. Les données s'y propagent suivant un mécanisme de FIFO.

  Le descripteur fd[0] est ouvert en *lecture* et le descripteur fd[1] est ouvert en *écriture*.

- Des processus ayant un lien de parenté peuvent communiquer à travers ce "tube" au moyen des fonctions *read* et *write*



http://unixwiz.net/techtips/remap-pipe-fds.html

Attention, rappelons que chaque processus fils a sa propre copie des descripteurs de fichiers de son père. Pour que des processus fils d'un même processus puissent communiquer par un tube commun (donc connaître les descripteurs de fichiers correspondants), ou pour qu'un fils puisse communiquer avec son père par le tube, il est impératif que le tube soit créé *avant* les processus fils.

- Par ailleurs, les "extrémités" inutilisées doivent être fermées (au moyen de la fonction *close*).

  En particulier, tous les "tubes" doivent être fermés par tous les processus lorsqu'ils ont fini leur traitement.

  Attention, un processus en attente de lecture sur un "tube" ne pourra "mourir" que s'il n'existe plus de processus capable d'écrire dans ce "tube" (i.e. descripteur de sortie ouvert).

- Valeur de retour de la fonction *pipe* :
  - 0 si succès,
  - sinon -1 (et errno est positionné).

- Exemple :

```
int main(void){
  int f[2];
  int x;
  pipe(f);
  switch (fork()){
     case -1 :
        perror("fork");
        exit(-1);
     case 0 :
        printf("Fils, numero %d\n", getpid());
```



```
        close(f[0]);
        printf("Entier saisi ?\n");
        scanf("%d",&x);
        while (x != 0){
           write(f[1], &x, sizeof(int));
           printf("Entier suivant ?\n");
           scanf("%d",&x);
        }
        close(f[1]);
     default :
        close(f[1]);
        while (read(f[0], &x, sizeof(int))>0)
           printf("Entier lu dans le tube : %d\n", x);
        close(f[0]);
        wait(NULL);
  }
  return 0;
}
```

VII.10 Redirections d'E/S

- La fonction *dup*

  int dup(int fd);

  permet de dupliquer un descripteur de fichier.

  L'argument correspond à un descripteur de fichier, et le nouveau descripteur créé prend *le plus petit numéro disponible dans la table des descripteurs*. Il a les points communs suivants avec le descripteur initial :
  - référence le même objet,
  - a le même mode d'accès,
  - utilise le même "pointeur".

- Remarques importantes :
  - l'application de la fonction *close* au descripteur initial n'aura pas d'incidence sur le nouveau descripteur.
  - la fonction *dup* sera notamment utilisée pour effectuer la **redirection de l'entrée standard et de la sortie standard** dans des processus qui exécutent (par *exec*) des commandes communiquant par "tube", comme illustré par l'exemple ci-dessous.

- Valeur de retour de la fonction *dup* :
  - le nouveau descripteur si succès,
  - sinon -1 (et errno est positionné).

- Exemple :

```
int main(){        // Cas "ps aux | grep sh"
  int f[2];
  pipe(f);
  switch (fork()){
     case -1 :
        perror("fork"); exit(-1);
     case 0 :
       printf("Fils 1. numero %d\n", getpid());
       // redirection de la sortie standard sur le tube
       close(1);
       dup(f[1]);  // prend le numéro 1
       close(f[1]);
       close(f[0]);   // fermeture descripteur inutilisé
       execlp("ps","ps","aux",(char *)0);
       // la sortie de ps ira dans le tube.
     default :
       switch (fork()){
          case -1 :
             perror("fork");
             exit(-1);
          case 0 :
             printf("Fils 2. numero %d\n", getpid());
             // redirection de l'entrée standard sur le tube
             close(0);
             dup(f[0]);  // prend le numéro 0
             close(f[0]);
             close(f[1]);  // fermeture descripteur inutilisé
             execlp("grep","grep","sh",(char *)0);
             // grep travaillera ainsi sur la sortie produite
             // par ps.
          default :
             // fermeture descripteurs inutilisés
             close(f[0]);
             close(f[1]);
             // attente mort des fils
             while (wait(NULL)!=-1);
       }
    }
    return 0;
}
```


VII.11 Complément – Fonction *stat*

- A chaque fichier est associé un i-nœud (ou "i-node") qui contient les informations relatives à ce fichier.

  La fonction *stat* permet d'aller récupérer ces informations :

  *int stat(const char *path, struct stat *buf);*

  où :
  - *path* est le nom du fichier,
  - *buf* est un pointeur sur la structure où se trouvent les informations.

- Parmi les champs de la structure *struct stat* se trouvent notamment :

```
ino_t     st_ino;    /* the file serial number */
mode_t    st_mode;   /* file mode */
nlink_t   st_nlink;  /* number of hard links to the file */
uid_t     st_uid;    /* user ID of owner */
gid_t     st_gid;    /* group ID of owner */
off_t     st_size;   /* total size of file, in bytes */
time_t    st_atime;  /* file last access time */
time_t    st_mtime;  /* file last modify time */
time_t    st_ctime;   /* file last status change time */
```

  Remarque : inclure <sys/stat.h> et <sys/types.h>

- Des macros et des masques permettent d'extraire des informations du champ *st_mode* :
  - Macros :

    S_ISDIR(m), S_ISREG(m), S_ISCHR(m) : le fichier est un répertoire, un fichier ordinaire, un fichier spécial de type caractère.
  - Masques :

    S_IRUSR, S_IWUSR, S_IXUSR (S_IRGRP, S_IWGRP, S_IXGRP, ou S_IROTH, S_IWOTH, S_IXOTH) : le propriétaire (le groupe, ou les autres) a le droit de lecture, d'écriture, d'exécution.

ANNEXE - PROGRAMMATION EN SHELL

1. Les scripts

- Les Shells offrent de puissants langages de programmation.

  Un *script* est un fichier contenant des commandes, il est écrit en utilisant le jeu d'instructions du Shell correspondant, et peut invoquer toute commande Unix.
  Il peut faire intervenir des *variables*, dont les contenus sont considérés par défaut comme des chaînes de caractères.

- On peut, depuis un Shell interactif quelconque, invoquer des scripts écrits dans d'autres Shells.

  Pour cela, il suffit (et d'ailleurs il est toujours préférable) que la première ligne du fichier soit de la forme :

  #!chemin_absolu_du_Shell_utilisé

  Exemples :
  ```
  #!/bin/bash
  #!/usr/local/bin/tcsh
  ```

- Le script sera exécuté en l'invoquant par son nom (éventuellement suivi de paramètres).
  L'exécution du script sera effectuée par un sous-shell.
  Pour que cela soit possible, il faudra lui donner la permission d'exécution.

  Exemple :
  ```
  $ emacs mon_script
  ... édition du script ...
  $ chmod u+x mon_script
  ```


- Variables spéciales :

	en (ba)sh	en csh
1. paramètres du script :	`$1,..., $9` (utilisation de *shift*)	`$1,..., $9, $10,...` ou `$argv[1],$argv[2],...`
2. nom du script :	`$0`	`$0`
3. nombre de paramètres :	`$#`	`$#argv`
4. liste des paramètres :	`$*`	`$*` ou `$argv[*]`
5. numéro de processus (utilisé en particulier pour générer des noms de fichiers temporaires) :	`$$`	`$$`

- Lecture au clavier :

en (ba)sh	en csh
`read mavariable`	`set mavariable=$<`

- Utilisation du résultat d'une commande (par exemple dans une affectation) :

en (ba)sh	en csh
`res=`ls | wc``	`set res=`ls | wc``

PETIT EXEMPLE

```
#!/bin/bash

echo "Je suis le script " $0 ", mon numero de process est " $$
echo "et on m'a passe " $# " parametres"
echo "Les trois premiers parametres : " $1 $2 $3
```

```
shift
echo "Les parametres 2, 3 et 4 sont : " $1 $2 $3
```

Exécution de ce script :

```
$ monscript un deux trois quatre cinq
Je suis le script monscript, mon numero de process est 1387
et on m'a passe 5 parametres
Les trois premiers parametres : un deux trois
Les parametres 2, 3 et 4 sont : deux trois quatre
```

2. Instructions conditionnelles

- L'instruction *if* :

en (ba)sh	en csh
if comm_cond *then* comm$_1$ *else* comm$_2$ *fi*	*if* (expr) comm
ou	*ou*

if comm_cond$_1$
then comm$_1$
elif comm_cond$_2$
then comm$_2$
…
fi
	if (expr$_1$) *then*
	comm$_1$
	else if (expr$_2$) *then*
	comm$_2$
	…
	endif

En bash, les commandes se trouvant après le *then* sont exécutées si la commande condition renvoie un code de retour nul (i.e. succès).

Cette commande condition peut faire appel à la commande *test* qui permet de :

- tester des caractéristiques de fichiers,
- comparer des chaînes de caractères ou des nombres.

En Csh, les commandes se trouvant après le *then* sont exécutées si l'expression condition est vraie. Cette expression doit être placée entre

parenthèses et est exprimée en termes d'opérateurs C, ou =~, !~

- La commande *test* (bash) :

La commande *test* est utilisée dans des instructions *if*, elle permet de tester la valeur d'une expression et renvoie un code de retour nul si l'expression est vraie et non nul sinon.

Syntaxe : *test* expression
ou
[expression *]*

Les expressions les plus courantes sont de la forme :

-f nomfic	vraie si le fichier *nomfic* existe
-d nomfic	vraie si *nomfic* est un répertoire
-r nomfic	vraie si *nomfic* existe et est accessible en lecture
-w nomfic	vraie si *nomfic* existe et est accessible en écriture
-x nomfic	vraie si *nomfic* existe et est exécutable
-n chaîne	vraie si la *chaîne* est non vide
s1 = s2	vraie si les deux chaînes sont identiques
s1 != s2	vraie si les deux chaînes sont différentes
n1 -eq n2	vraie si les deux entiers sont égaux
n1 -ne n2	vraie si les deux entiers sont différents
	(sont aussi utilisés -lt, -le, -gt, -ge)

les opérateurs -o, -a et ! peuvent aussi être utilisés.

- Exemple :

```
#!/bin/bash
if [ $1 = "data"  -a  -r $1 ]
then  if grep -q '[0-9]' $1
      then echo $1 " contient des chiffres"
      else echo "pas de chiffres dans " $1
      fi
else echo $1 " pas le bon fichier ou ne peut pas etre lu"
fi
```

```
#!/bin/csh
if ($1 == "data"  &&  -r $1) then
    if { grep -q '[0-9]' $1 } then
    echo $1 " contient des chiffres"
    else
    echo "pas de chiffres dans " $1
    endif
else
echo $1 " pas le bon fichier ou ne peut pas etre lu"
endif
```

- L'instruction *case* ou *switch* :

en (ba)sh	en csh
case chaîne *in* motif$_1$) comm$_1$;; motif$_2$) comm$_2$;; … motif$_k$) comm$_k$;; *esac*	*switch* (chaîne) *case* motif$_1$: comm$_1$; *breaksw* … *default:* comm$_k$ *endsw*

La *chaîne* peut être une lettre ou un mot, ou même un nombre, ou faire intervenir des caractères spéciaux du shell. Elle peut résulter de l'exécution d'une commande, au moyen de `…`

En bash, le shell recherche le premier motif qui correspond à *chaîne*, et exécute la ou les commandes correspondantes. Plusieurs motifs peuvent être regroupés dans une même alternative, en utilisant |. Le motif * correspond au *default* de la version Csh.

En Csh, l'instruction *switch* se comporte de façon similaire au *switch* de C, la commande *breaksw* est nécessaire pour sortir d'une alternative.

- Exemple :

```
#!/bin/sh
case $# in
0) echo "pas de parametres..." ;;
 1 | 2) echo "on a passe 1 ou 2 parametre(s)..." ;;
*) echo "on a passe plus de 2 parametres..." ;;
esac
```



```
#!/bin/csh
switch ($#argv)
case 0:
   echo "pas de parametres..." ; breaksw
 case 1:
case 2:
   echo "on a passe 1 ou 2 parametre(s)..."
   breaksw
default:
   echo "on a passe plus de 2 parametres..."
endsw
```

3. Les boucles

- L'instruction *for* ou *foreach* :

en (ba)sh	en csh
for var *in* chaîne$_1$ … chaîne$_k$ *do* commandes *done* *ou* *for* var *do* commandes *done*	*foreach* var (liste_chaînes) commandes *end*

En bash, les commandes sont exécutées pour toutes les valeurs possibles de *var*. Soit on donne une liste de valeurs possibles *chaîne$_1$ … chaîne$_k$* , soit on remplace cette liste par *, ce qui a pour effet de considérer comme valeurs possibles les noms des fichiers du répertoire courant. Enfin, la deuxième forme a pour effet de considérer comme valeurs possibles les paramètres du script.

En Csh, les commandes sont exécutées pour toutes les valeurs possibles de *var*, qui sont listées dans *liste_chaînes*.

- Exemple :

```
#!/bin/sh
for nomfic in *
do
    if [ -f $nomfic ]
    then echo $nomfic " : fichier ordinaire"
    elif [ -d $nomfic ]
    then echo $nomfic " : repertoire"
    else echo $nomfic " : autre type de fichier"
    fi
done
```

```
#!/bin/csh
foreach nomfic (`ls`)
    if ( -f $nomfic ) then
    echo $nomfic " : fichier ordinaire"
    else if ( -d $nomfic ) then
         echo $nomfic " : repertoire"
         else echo $nomfic " : autre type de fichier"
         endif
    endif
end
```

- L'instruction *while* :

en (ba)sh	en csh
while comm$_1$ *do* comm$_2$ *done*	*while* (expr) commandes *end*

En bash, la ou les commandes *comm$_2$* sont exécutées répétitivement tant que la commande (ou la dernière commande de la suite de commandes) *comm$_1$* renvoie un code de retour nul (i.e. succès).

En Csh, les *commandes* sont exécutées répétitivement tant que l'expression *expr* est vraie. La syntaxe de cette expression est similaire à la syntaxe des expressions utilisées dans les instructions *if*.

- Exemple :

```
#!/bin/sh
echo "Fichier a traiter ?" ;  read rep
while [ ! -f $rep ]
do echo "Le nom n'est pas bon" ; read rep
done
```

```
#!/bin/csh
echo "Fichier a traiter ?" ;  set rep=$<
while (! -f $rep)
echo "Le nom n'est pas bon";  set rep=$<
end
```


4. Les scripts récursifs

- En bash comme en Csh, on peut concevoir des scripts récursifs.
  Ceci est utile lorsqu'un même traitement doit être appliqué plusieurs fois, notamment lors d'un parcours de la sous-arborescence d'un répertoire avec traitement de chaque nœud.

- Attention, comme toute procédure récursive, un tel script doit prévoir (au moins) un cas d'arrêt de la récursion.
  Dans le cas d'un parcours de la sous-arborescence d'un répertoire avec traitement de chaque nœud, le cas d'arrêt typique est l'arrivée sur un fichier non répertoire.

- Exemple : commande *mycp*

```
#!/bin/sh
if [ -f $1 ]
# fichier ordinaire : copie normale
then cp $1 $2
# répertoire : appel récursif pour tous ses éléments
else mkdir $2
     for nomfic in `ls $1`
        do mycp $1/$nomfic $2/$nomfic
     done
fi
```

```
#!/bin/csh
if ( -f $1 ) then
cp $1 $2
else mkdir $2
     foreach nomfic (`ls $1`)
        mycp $1/$nomfic $2/$nomfic
     end
endif
```