UNIVERSITE MOHAMMED V – AGDAL
Faculté des Sciences
Rabat

2008

Les bases d'Unix

(3ème année de licence)

Filières fondamentale et professionnelle

Pr. M. JEDRA

Table des matières

UNIX
Présentation générale
Unix est une marque déposée de AT & T

1.1 Historique:

1968: Fin du projet MULTICS,
Etude commune des Bell Labs et General Electric au MIT.
Axes de recherche:
- mémoire virtuelle,
- protection,
- ordonnancement,
- système de gestion de fichiers.

Bell Labs sont des labos de AT &T et Westen Electric.

1970: Première version d'Unix
- système mono-utilisateur,
- système de gestion de fichiers,
- outils de traitement de textes,
- noyau de système élémentaire,
- interpreteur de commandes élémentaire.

Travail initialisé par Ken THOMPSON

1971: Première version d'Unix
+ documentation
+ plusieurs extensions
- système de fichiers,
- gestion de processus,
- interface système,
- utilitaires,
- transport sur PDP 11/20

Première version officielle interne Unix V1 signée:
Ken THOMPSON et Denis RITCHIE.

1972: Unix V1+ "pipes" = V2
Transport sur PDP 11/20/30/34/40/45/60/70.

1973: Unix ré-écrit en C
BCPL ==>B ==>NB ==> C (D. RITCHIE)
<Langages interprétés> <Compilateur cc >

1974: Unix V5, distribuée gratuitement à des universités
(UC Berkley, Colombia U.)

1975: Unix V6, Première version distribuée pour 250 $.

1977: Unix 32V sur Vax (32bits)
UC.Berkley (BSD) ==> C-Shell, éditeurs,...
BSD: Berkley Software Distribution

1979: Unix V7 = V6+portabilité
plusieurs implantations.

1980: "ONYX" première version pour micros

1981: Première version commercialisée par AT & T ==> System III
U.C.Berkley ==> BSD4.1
Deux produits indépendants

1983: AT et T ==> System V
U.C.Berkley ==> BSD4.2

1984: Microsoft ==> Xenix version pour PC 8086

1987: AT et T ==> System V Release 3
U.C.Berkley ==> BSD4.3
Santa Cruz Opérations ==> SCO Unix pour PC 80386

1990: AIX3.0 ==> Le système Unix d'IBM

1991: AT & T et SUN ==> System V Release 4
Intégration des fonctionnalités du système V et BSD
Standard (Posix)

Actuellement: AIX d'IBM, Solaris de SUN, Linux, ..., etc

1.2 Structure du système

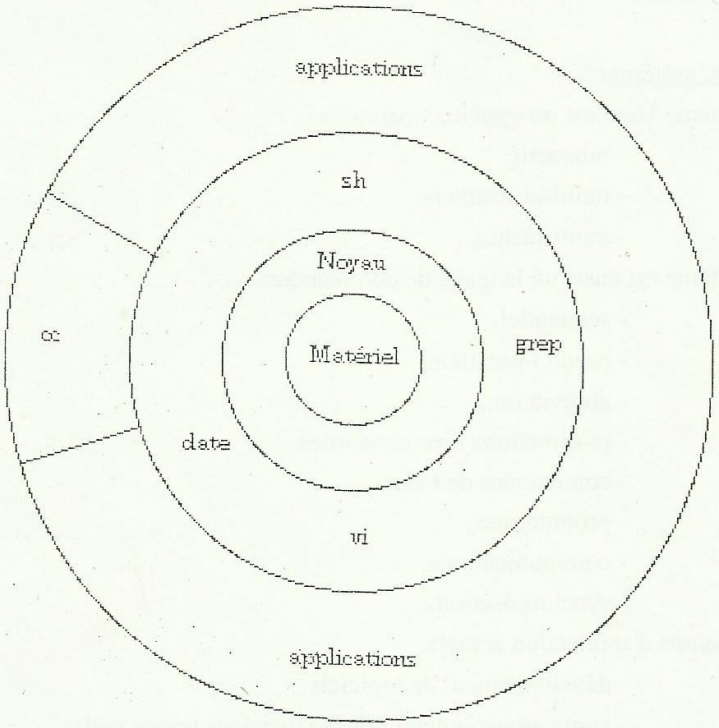


Fig 1.1 Structure du système

Le noyau :

- gère les ressources matérielles (mémoires, unités d'E/S...), les fichiers, l'allocation du temps CPU.
- gère les processus.

Le shell :

Le shell (coquille) est un utilitaire qui sert d'intermédiaire entre la saisie des commandes et le noyau, c'est un interpréteur de commandes. Il existe plusieurs shell:

sh	Bourne-shell (standard)	Steve Bourne 1970.
csh	C-shell	(Berkley)
ksh	Korn-shell	David Korn
bach	Bourne-again-shell (Linux)	
...		

Le ksh est une extension du Bourne-shell qui intègre les caractéristiques principales du C-shell.

La couche des outils et des applications:

C'est l'ensemble des programmes et des utilitaires qui ont été écrits petit à petit pour améliorer le système.

Les qualités du système:

Le système Unix est un système opératoire:

- interactif,
- multi-utilisateurs,
- multi-tâches,

Le système est aussi un langage de commandes:

- séquentiel,
- pseudo-parallèle,
- abréviations,
- re-directions d'entrée-sorties
- commandes de base,
- programmes,
- communications,
- synchronisation...

Les champs d'application actuels:

- développement de logiciels
- applications industrielles (extensions temps-réel)
- communications (courrier électronique, transfert de fichiers,...,etc)

Le système évolue maintenant vers la distribution et les systèmes répartis

- système de gestion de fichiers répartis (NFS de Sun),
- exécution de commandes à distance,
- extension des mécanismes de nommage,
- noyau à commutation de messages,
- communications inter-processus,

L'environnement de programmation:

- langages: C, C++, Pascal, Fortran, Lisp, APL, Prolog, JAVA,...,
- éditeurs de texte ligne (ed), écran (vi, emacs,...,etc),
- utilitaires de traitement de texte (TROFF, NROFF),
- communications inter-utilisateurs (mail),
- outils graphiques de base (GKS, PHIGS),
- interfaces utilisateur multi-fenêtres (Xwindows, Sunview, KDE ...,etc),
- outils de gestion de logiciels (MAKE, SCCS),
- outils pour Internet (TCP/IP, DNS, POP...etc)
- service d'authentification Client-Serveur (Kerberos)
- applications...

1.3 Normalisation

Le but de la normalisation est de définir les interfaces avec le système et de laisser libre la mise en oeuvre du noyau (structure, optimisation,...,etc).

Les organismes internationaux de normalisation:

ISO : International Standards Organisation

JTC1=Join Technical Comitee (comité technique dédié à l'informatique)

Les organismes nationaux:

ANSI (USA), BSI (UK), DIN (RFA), AFNOR (F),...,etc

Les sociétés de service informatique:

Les SSI font des propositions de normes, et font des choix pour rendre les normes opérationnelles.

/usr/group	USA	"USR/group standards" 1984
IEEE	USA	"POSIX: Portable Open System Interface"
X-OPEN	EU	"Portability Guide"

Les sociétés:

Imposent de nouveaux standards

AT & T (SIVD = System V Interface Definition)

AT & T (SVVD = System V Vérification Suite)

Les groupements:

Unix International : AT & T, SUN

OSF: IBM, HP, DEC, BULL,...,etc

IEEE DP12 du comité 1003 chargé de POSIX

Le comité 1003 se compose de sous-comités:

1003.1: Norme de base.

1003.2: Langages de commande (shell).

1003.3: Tests et suite de vérification de conformité.

1003.4: Aspect temps réel.

1003.5: POSIX et ADA

1003.6: Sécurité à partir de la classification de l'Orange Book du DOD.

Cette norme POSIX assure la portabilité du système indépendamment du langage C.

X-OPEN groupement de constructeurs

- Des constructeurs européens (BULL, ICL, Olivetti, Siemens-Nixdorf...)
- Quelques constructeurs Américains dont AT & T

Le but de X-OPEN est de proposer, d'influencer et d'adopter des normes de manière cohérente dans un parc hétérogène.

UNIX
Communication avec le système

2.1 Session utilisateur

```
   login: nom-utilisateur < rc >
password: xxxxxx < rc >          # mot de passe non affichée.
```

Si soit le nom-utilisateur ou le mot de passe est incorrect, le système répondra par:

```
login incorrect
login:
```

suivi de `password:`

Si votre `login` est correct automatiquement il y a apparition du prompt du shell :

`$` avec le Bourne shell
ou `%` avec le C–shell
ou un autre prompt choisi par l'administrateur du système.

Pour changer le mot de passe on utilise la commande `passwd`

```
$passwd
Changing password
Old password:  [Entrer l'ancien mot de passe]
New password:  [Entrer le nouveau mot de passe]
Re-enter new password: [Entrer le nouveau mot de passe]
```

On signale que des fichiers prédéfinis contenant des commandes sont exécutés au début et à la fin d'une session, par exemple : des commandes de configuration du terminal, de lancement d'une application,...,etc.

Pour BSD / C-shell : `.login` et `.logout`
Pour AT & T / shell (Bourne) : `.profile`

Pour sortir de la session on utilise :

```
$logout  < rc >          # version BSD
< Ctrl +D >              # version AT &T
$exit <rc>               # Linux
```

2.2 Communication entre utilisateurs

La communication entre utilisateurs peut se faire en ligne ou en différé.

Communiquer en ligne (`write` et `wall`)
- communication immédiate avec les autres utilisateurs "logués".
- une réponse immédiate est possible

Communiquer en différé (`mail`)
- le message est permanent, il est sauvé dans un fichier.
- différentes possibilités lors de l'obtention d'un message

`wall` envoie un message à tous les utilisateurs. L'appel de cette commande suppose des droits d'accès adéquats (administrateur). Le plus souvent elle est dans le répertoire `/etc`.

Exemple :

```
$wall
Le système sera arrêté dans 10 minutes.
Salutations  Administrateur
$
```

Le message apparaît sur tous les terminaux des utilisateurs "logués".

`write` permet d'envoyer des messages en ligne à un autre utilisateur connecté.

```
$write utilisateur  [Terminal]
 message
 .....
```

Le terminal est spécifié dans le cas ou l'utilisateur est connecté à travers plusieurs terminaux. Dans le message reçu par le destinataire, se trouvent les informations suivantes :

- le nom de l'expéditeur,
- le terminal d'émission,
- l'heure de début de l'émission,

....etc.

Exemple :

Terminal tty01 (Youssef)	Terminal tty02 (Karim)
`$write` Karim	
	Message from Youssef tty01.....
	`write` Youssef
Message from Karim tty02.....	
Bonjour Karim	
	Bonjour Karim
Veux tu aller au cinéma ce soir	
	Oui
Oui	
OK	
`<Ctrl+D>`	OK
$	EOT
	`<Ctrl+D>`
	$

`<Ctrl+D>` permet de sortir de `write`

Pour envoyer un message à un utilisateur, celui-ci doit en donner les droits avec la commande de contrôle de réception de message `mesg`

```
$mesg [y ou n]
```

Elle permet à l'utilisateur de verrouiller son écran face aux éventuels messages.
`y` (oui) pour annuler le verrouillage `n` (non) pour établir le verrouillage

mail permet d'échanger du courrier avec d'autres utilisateurs. En général les messages arrivés atterrissent dans un fichier du répertoire /usr/mail (ou usr/spool/mail).
Une fois les messages lus ils sont soit stockés dans des fichiers en local ou détruits.
Par défaut, il s'agit du fichier mbox, dans le répertoire courant de l'utilisateur.

```
$mail                          # entrée en mode commande,
message1                       # les messages sont visualisés l'un après l'autre.
? <rc>                         # le prompt est généralement ? ou &
message2
?
```

Après le prompt ? on peut utiliser l'une des commandes suivantes :

- **d** supprimer le message et afficher le suivant
- **s [fichier]** sauvegarder le message dans mbox, ou dans [fichier] si spécifié.
- **-** message précédent.
- **r** réponse à l'expéditeur du message lu
- **w** identique à s mais sans entête
- **h** liste les entêtes des messages
- **m [utilisateurs]** renvoie du message aux [utilisateurs]
- **q** sortie de mail avec sauvegarde dans mbox
- **x** sortie sans sauvegarde
- **?** aide en ligne

```
$mail [adresse]
Subject:                #Sujet du message
Cc:                     #Copie conforme
message
« . » ou <Ctrl+D>
```

Exemples

```
$mail Ahmed                    # adresse dans la même machine
Subject : Sport
Cc : Youssef
Il y aura une réunion concernant le sport universitaire
le Lundi 09/10/02 à 18h à la salle des séminaires.
Salutations Karim
.

$mail
From Karim Thu Oct 02 16 :43 :27.....
.....
Il y aura une réunion concernant le sport universitaire
le Lundi 09/10/02 à 18h à la salle des séminaires.
Salutations Karim
?

$mail Ahmed < message          # envoie d'un message sous forme de fichier

$mail Ahmed@fsr.ac.ma < message   # envoie d'un message à un utilisateur sur
                                  # une autre machine
```

2.3 Programmes d'information

Ce sont des commandes qui servent à l'utilisateur pour obtenir des informations sur l'état de son système Unix. Il s'agit, entre autres, de :

`date` commande qui affiche la date et l'heure du système.

```
$date [Format]
$date mmddHHMM[yy]          # mm le mois dd le jour
                            # HH l'heure MM les minutes
                            # yy l'année
```

`who` commande qui indique les utilisateurs connectés au système.

```
$who [option]
$who am I
```

Cette commande sans paramètres affiche pour tous les utilisateurs connectés les informations suivantes :

- nom de l'utilisateur,
- terminal sur lequel il travaille,
- heure de connexion,
-...., etc

`cal` affiche le calendrier

```
$cal [[mois] année]
```

`man` affiche la documentation intégrée concernant une commande.

```
$man commande
```

Le système Unix possède une documentation intégrée de toutes les commandes, c'est une aide (Help) mis à la disposition de l'utilisateur.

UNIX
Le système de fichiers

3.1 Les fichiers Unix

Fichier Unix

C'est un ensemble d'informations associées à un nom. Il est considéré comme une suite d'octets (caractère ASCII on code machine) stockés sur disque ou bande magnétique. Il n'y a pas d'organisation spéciale (séquentielle, indexée,...)

Les types de fichiers Unix :

- Fichiers ordinaires (segments):
Un fichier ordinaire est une séquence de caractères (sources, binaires, textes). Pas de structure imposée par le système; la structure logique des fichiers (enregistrements) est fixée par les programmes d'applications.

- Répertoires (directories) :
Un répertoire est un catalogue qui contient les noms des fichiers et des sous répertoires.

- Fichiers spéciaux (unités d'E/S) :
Ces fichiers représentent des interfaces avec les périphériques gérés par le système d'exploitation.

Organisation utilisateur du système de fichiers :

C'est une structure arborescente qui possède un répertoire racine et dont les noeuds sont des répertoires. Chaque répertoire peut contenir des fichiers et d'autres répertoires.

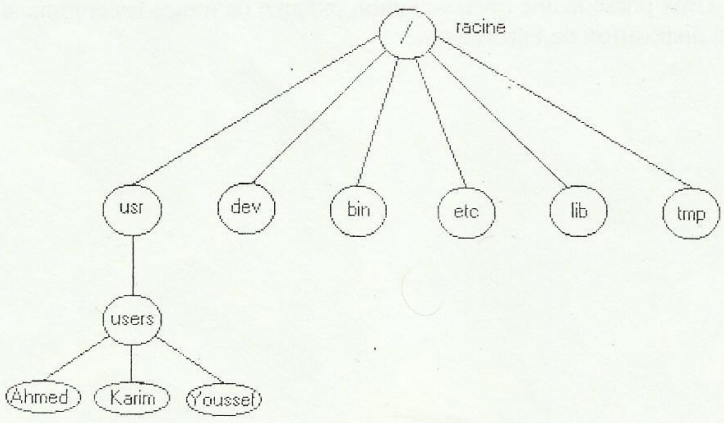


Fig 3.1 Schéma simplifié du système de fichiers

`usr` contient les répertoires utilisateurs (`home` pour Linux).
`lib` contient les librairies (par exemple lib.c)
`dev` contient les fichiers spéciaux : `tty01, lp01, dk01, fl01`,...,etc

`bin` contient les commandes : **`date`**, **`cat`**, **`who`**,...,etc
`etc` contient les fichiers système : `passwd`, `group`, `hosts`,...,etc
`tmp` contient les fichiers temporaires du système.

Tout fichier est nommé par son chemin d'accès (pathname) dans l'arborescence.

`/usr/users/Ahmed/mbox` est la boite aux lettres de l'utilisateur Ahmed.

`/usr/users/Ahmed` est le répertoire personnel (Home directory) de l'utilisateur Ahmed.

Conventions de nommage

`..` répertoire parent
`.` répertoire courant
`/` racine de l'arborescence.

Le nom d'un répertoire ou d'un fichier peut contenir des chiffres et des lettres (minuscules ou majuscules), ainsi que des caractères spéciaux dont l'utilisation est déconseillée.

Organisation physique

Le disque dur est un ensemble de plateaux empilés verticalement sur un même axe. Chaque plateau est composé d'une ou de deux faces, divisées en pistes. Chaque piste est elle-même divisée en secteurs de taille fixe entre 512 à 4096 octets. Le secteur constitue la plus petite unité de lecture/écriture sur le disque. L'opération qui permet d'organiser un disque vierge en un ensemble de pistes et de secteurs s'appelle formatage physique.

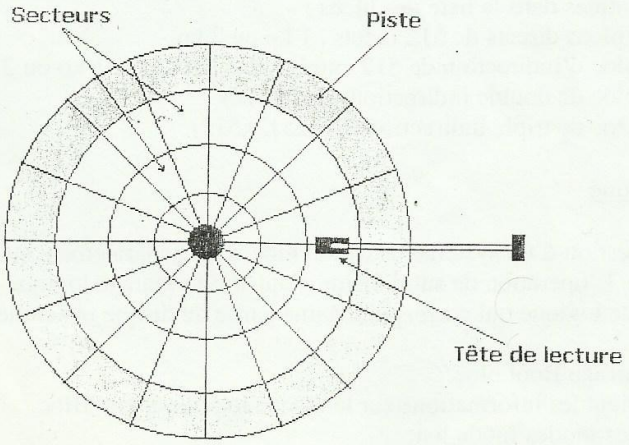


Fig. 3.2 Organisation d'un disque dur

Le système d'exploitation utilise pour les échanges entre la mémoire centrale et le disque dur des blocs et non des secteurs. Un bloc est constitué de plusieurs secteurs. Les méthodes d'allocation des blocs aux fichiers sont nombreuses mais la technique la plus utilisée est l'allocation indexée.

Allocation indexée

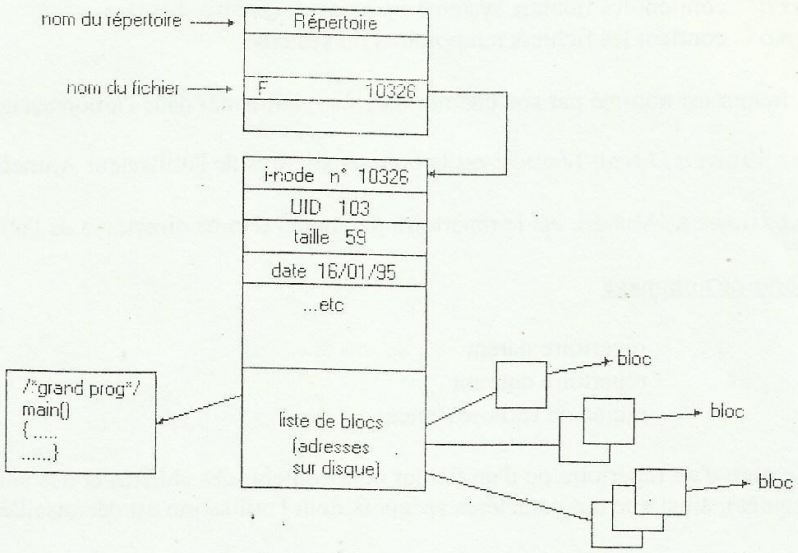


Fig. 3.3 Organisation interne d'un fichier

Un fichier : i_node --------> numéro de bloc
Un bloc (512 , 1024 ou 2048 octets): taille, UID, liens, date, liste de blocs utilisés...etc
Indirections ----------> arbre de blocs

Il y a 13 entrées dans la liste des blocs :
10 blocs directs de 512 octets , 1 ko ou 2 ko.
1 bloc d'indirection de 512 entrées de 512 octets , 1 ko ou 2 ko.
1 bloc de double indirection (512x512)
1 bloc de triple indirection (512x512x512)

Organisation logique

Pour faciliter la gestion d'un système d'exploitation il est généralement subdivisé en morceaux appelés partitions. L'opération de subdivision s'appelle formatage logique. Chaque partition constitue un disque logique qui correspond à une partie du disque physique et elle comporte :

- un bloc de démarrage Boot bloc,
- un bloc qui contient les informations sur le disque logique Super Bloc,
- une table pour les inodes Inode list,
- des blocs de données.

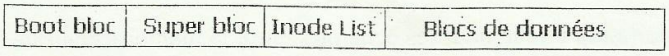


Fig. 3.4 Structure d'une partition

3.2 Commandes de manipulation de fichiers

```
$ls [options] [nom de répertoire]
```

La commande `ls` affiche les noms des fichiers et des répertoires du répertoire indiqué par son nom (chemin). Si le nom du répertoire est omis c'est le contenu du répertoire courant qui est affiché.

```
[options]
  -l        + les droits d'accès
  -c        en colonnes
  -a        les fichiers commencent par . (cachés).
  -d        les répertoires
  -i        + les numéros i-node.
 ..... etc
```

Les options de `ls` peuvent être mixées par exemple : `$ls - ail`

`$cat F G`	affichage du contenu des fichiers **F** et **G**
`$cp F/G`	copie de **F** sur **G**.
`$rm F`	destruction de **F**.
`$mv F G`	re-nommage de **F** en **G** ou déplacement de **F** dans **G** si **G** est un répertoire.
`$more F`	équivalent de `cat` mais page par page.
`$ln F G`	deux noms pour un même fichier (même numéro d'i-node).
`$pr F`	impression paginée de F chaque page débute par une entête contenant : <date> <heure> <nom> <numéro de page>
`$lp F`	impression d'un fichier F

3.3 Commandes de manipulation des répertoires.

`$pwd`	impression du répertoire courant dans l'arborescence.
`$cd <nom_rep>`	changement de répertoire courant.
`$mkdir <nom_rep>`	création d'un sous répertoire.
`$rmdir <nom_rep>`	destruction d'un répertoire s'il est vide.

3.4 Redirections des entrées/sorties

Les entrées/sorties standards.

Quand un utilisateur se connecte à Unix, trois fichiers sont ouverts :
- entrée standard (**stdin**) : le clavier où sont entrées les commandes.
- sortie standard (**stdout**) : l'écran où sont affichées les résultats des commandes.
- sortie d'erreur (**stderr**) : l'écran où sont affichées les erreurs éventuelles.

Un nombre (le descripteur de fichier) est affecté à chaque fichier ouvert,

```
0 : stdin
1 : stdout
2 : stderr
```

Certains commandes n'écrivent pas le résultat de leur action sur la sortie standard, mais dans un fichier , par exemple la commande **mail**.

Certains commandes ne lisent pas sur l'entrée standard, mais sur un ou des fichiers. par exemple la commande **cat** .

Redirection des entrée/sorties

Le shell est en mesure de rediriger les résultats de l'action d'une commande vers un fichier diffèrent de la sortie standard (écran). Il peut aussi lire les données d'un fichier diffèrent de l'entrée standard (clavier) pour une commande qui attend les données du clavier.

<	redirection de l'entrée standard
>	redirection de la sortie standard (création)
>>	redirection de la sortie standard (ajout)
<<	redirection spéciale de l'entrée standard (Herescripts).
2>	redirection de la sortie d'erreurs (création)
2>>	redirection de la sortie d'erreurs (ajout à un fichier existant)

Exemples :

```
$ls
Etudiant
Professeur
Spline.cpp
$ls > F
$ls
Etudiant
Professeur
Spline.cpp
F
$cat F
Etudiant
professeur
Spline.cpp
$
```

```
$who > Connecte
$cat Connecte
Ahmed   tty01 Mai 12..... 10:18
Youssef tty02 Mai 12..... 08:45
$date >> Connecte
$cat Connecte
Ahmed   tty01 Mai 12..... 10:18
Youssef tty02 Mai 12..... 08:45
12 Mai 1995 12:15:34
$
```

```
$wc << fin
a b c d
e f g h
fin
   2    8   16
$
```

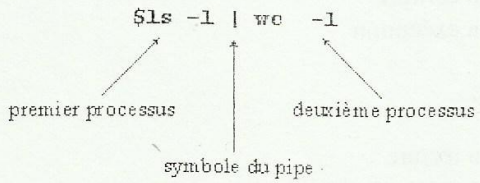
La commande **wc** compte les lignes, les mots et les caractères d'un texte y compris les blancs. Le mot fin est utilisée uniquement pour indiquer à la commande la fin du script (texte). On peut utiliser n'importe quel symbole ou chaîne de caractères .

```
$mail Ahmed < message      # envoie du fichier message à l'utilisateur Ahmed
```

```
$wc < F > G                # mise du résultat de wc dans G.
```

Notion de pipe:

La sortie d'un processus est directement connecté à l'entrée d'un autre. Ceci permet d'éviter de créer des fichiers temporaires, par exemple :



Logiquement ceci est équivalent à :

```
$ls -l > F      rediriger le résultat de ls -l dans un fichier F
$wc -l < F      puis appliquer wc -l sur ce fichier
```

mais

- pas de sérialisation des processus
- pas de fichier F utilisé
- **ls** et **wc** sont deux commandes (processus) s'exécutant en pseudo-parallèlisme sur le principe du producteur consommateur.

Remarques:

- Les pipes sont unidirectionnels
- Les pipes sont crées par le système
- La synchronisation est réalisé par le système

3.5 Les droits d'accès

Notion d'utilisateur

Tout utilisateur est enregistré dans deux fichiers systèmes :
`/etc/passwd` et `/etc/group`

`/etc/passwd` contient les informations suivantes pour chaque utilisateur :

- nom de login
- mot de passe crypté
- numéro unique d'utilisateur : UID
- numéro unique de groupe : GID
- nom complet de l'utilisateur
- répertoire initial
- interpréteur de commande

`/etc/group` contient les informations sur le groupe auquel appartient l'utilisateur :

- nom du groupe
- numéro unique de groupe
- liste des utilisateurs de groupe

Les droits d'accès d'un fichier déterminent qui peut accéder à ce fichier et comment. Ils sont définis à l'aide des informations contenues dans les fichiers systèmes `passwd` et `group`

Droits d'accès aux fichiers:

r	accès en lecture
w	accès en écriture
x	accès en exécution

Droits d'accès aux répertoires:

r	accès en lecture
w	accès en création, modification, destruction
x	accès au nom (se positionner dans le répertoire)

La commande `ls -l` donne la liste des fichiers du répertoire courant avec type et droits d'accès :

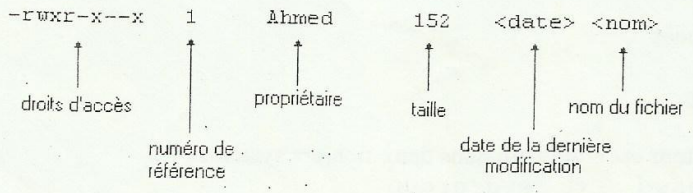
```
<type> <propriétaire> <groupe> <autres>
  -          rwx          r-x       --x
```

Il y a trois classes d'utilisateurs :

- **u** le propriétaire du fichier
- **g** le groupe du propriétaire
- **o** les autres utilisateurs

Exemples :

- fichier ordinaire



- fichier spécial en mode bloc (périphérique en mode bloc : disque)

```
brw-rw-rw-     1   root  ...  7   <date>  /dev/dk
```

- fichier spécial mode caractère (périphérique en mode caractère : transmission)

```
crw-rw-rw-     1   Ahmed  ...  214   <date>  /dev/tty1
```

- répertoire

```
drwxr-x--x     2   Ahmed  ...  593  <date>  nom_directory
```

Pour accéder à une feuille ou à un noeud dans une arborescence de répertoire, il faut avoir le droit en x sur tous les répertoires de niveau supérieur (c.à.d appartenant au chemin d'accès).

Les restrictions d'accès s'appliquent aux utilisateurs, un seul est exempt des contrôles d'accès : le super_utilisateur qui a pour login : **root** et le **UID= 0**

Changement de propriétaire et de groupe

Ahmed un utilisateur propriétaire de F (ou administrateur) il peut changer le propriétaire F à l'aide de la commande **chown**

```
$chown   F Karim
```

Il peut aussi changer le groupe avec **chgrp**

```
$chgrp  F    groupe_Karim
```

Modification des droits d'accès

Les droits d'accès peuvent être modifiés par la commande **chmod**

```
$chmod    u+x   F          # ajout du droit d'exécution à l'utilisateur.
$chmod    g+w   F          # ajout du droit d'écriture pour le groupe
$chmod    o=x   F          # droits des autres mis à x uniquement.
$chmod     +x   F          # ajout du droit x à tous
```

Une autre forme d'utilisation de **chmod** avec des chiffres de la base octale.

```
$chmod   Nombre en base 8      Fichier(s)
```

Pour cela, il faut affecter des chiffres en base 8 aux paramètres symboliques des droits d'accès individuels:

Propriétaire			Groupe			Autres		
r	w	x	r	w	x	r	w	x
400	200	100	40	20	10	4	2	1

Le nouveau droit d'accès est la somme des droits en octal.

Exemple:

```
$chmod  664    F
```

```
600 ----> droit d'accès du propriétaire  rw-
 60 ----> droit d'accès du groupe        rw-
  4 ----> droit d'accès des autres       r--
```

Droits d'accès par défaut

Dans le système Unix les droits d'accès maximales pour les fichiers normaux et les répertoires sont :

```
rwxrwxrwx (777) pour les répertoires
rw-rw-rw- (666) pour les fichiers
```

La commande **umask** permet de positionner les droits d'accès des fichiers par défaut.

Si aucun masque n'est positionné les droits par défaut sont:

```
Les éditeurs -----> -rw-r--r--   (644)
     mkdir -----> drwxr-xr-x   (755)
```

Exemple:

```
$umask 022
```

```
022 en octal: 000 010 010
 ou exclusif  111 111 111  (777)
              111 101 101  (755)  ----> drwxr-xr-x  pour les répertoires

022 en octal: 000 010 010
 ou exclusif  110 110 110  (666)
              110 100 100  (644)  ----> -rw-r--r--  pour les fichiers
                                        (éliminer les x)
```

`drwxr-xr-x` et `-rw-r--r--` sont devenus les valeurs par défaut.

3.6 Génération de noms (Jockers)

Le shell possède des caractères spéciaux pour définir des critères de recherche pour les noms de fichiers :

- `*` remplace une chaîne de longueur variable, même vide.
- `?` remplacé un caractère unique quelconque.
- `[...]` représente une série ou une plage de caractères.
- `[!...]` le fait de commencer la définition par ! inverse la recherche.

Exemples:

```
$ls  *.cpp        # liste de tous les noms de fichiers du répertoire courant
                    se terminant par .cpp

$ls [a-z]*        #liste des noms des fichiers commençant par une lettre
```

```
$ls   a*          #liste des noms des fichiers commençant par a.

$ls   [aA]*       #liste des noms des fichiers commençant par a ou A

$ls   ?*          #liste des noms de fichiers dont le nom est d'au moins un caractère

$ls   a??         #liste des noms des fichiers ne contenant que 3 caractères dont
                   la première est a
```

Pour annuler les fonctions spéciales des caractères spéciaux en utilise :

- backslash (\)
- guillemets ("...")
- apostrophes ('...')

Recherche de fichiers:

La recherche de fichiers se fait à l'aide de la commande **find**.

```
$find <nom_directory> critère option_commande
```

`nom_directory` ----> point de départ de la recherche.

`critère` ----> composé des opérateurs suivants :

- `-name <nom_fichier>` recherche par nom de fichier
- `-user < nom_utilisateur>` recherche par propriétaire
- `-group <nom_groupe>` recherche par groupe
- `-type <caractère>` recherche par type indiqué (d,b,c,...)
- `-size <n>` recherche par taille de **n** blocs
- `-inum <n>` recherche par numéro i_node **n**
- `-atime <n>` recherche de fichier modifié depuis **n** jours
- `-perm <n>` recherche par droit d'accès **n** en octal

`option_commande` ----> commande appliquée au fichier trouvé :

- `-print` affichage du chemin du fichier
- `-exec <commande> { } \;` exécution de **commande** sur le fichier indiqué par { }. la commande doit terminer par un \;
- `-ok <commande> { } \;` idem `-exec` en mode conversationnel.

Exemples:

```
$find . -name  *.cpp  -print      # afficher les noms des fichiers
                                  sources en cpp du répertoire courant
$find . -size 10 -print           # afficher les fichiers de 10 blocs

$find . -type b -size 10 -print   # idem de type b ayant 10 blocs

$find . -size 0 -exec rm {}\;     # enlever les fichiers vides

$find . -type d -ok ls -l {}\;    # afficher le contenu des sous
                                  répertoires en conversationnel
```

UNIX
L'éditeur pleine page vi

4.1 Les modes de travail de vi

vi est un éditeur pleine page développé par l'Université de Californie Berkeley. **vi** est basé sur un éditeur "ligne" sous-jacent **ex** qui lui-même intègre l'éditeur ligne d'Unix **ed**. Il travaille selon deux modes:

- **Mode commande**, dans lequel toutes les saisies, lettres, touches et combinaisons de touches, représentent des commandes. En général , ces commandes ne terminent pas par la touche Entrée.

- **Mode saisie**, dans lequel toutes les saisies sont enregistrées dans le fichier sous format texte.

- **Mode ligne de commande** (mode **ex** (**ed**)), dans lequel il est possible d'utiliser des commandes spéciales, redirigées depuis l'éditeur **ex**. Ces commandes demande, en final, une validation par la touche Entrée.

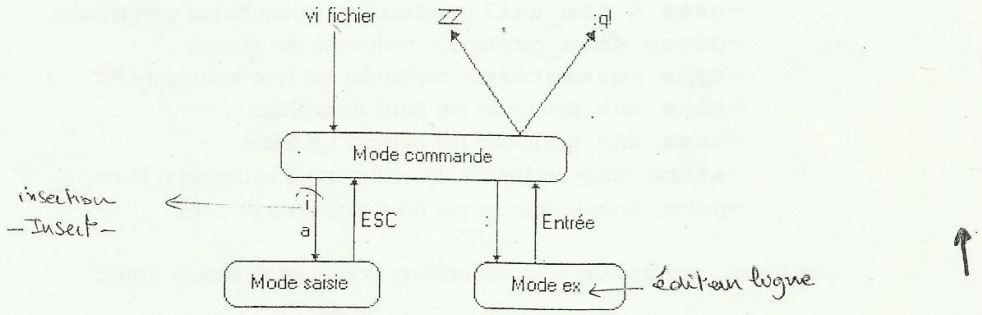


Fig. 4.1 Passages entre les modes de travail de **vi**

4.2 Edition d'un fichier

```
$vi [option] fichier
```

vi prend en argument le nom du fichier à éditer. Si le fichier n'existe pas, il est automatiquement créé. **vi** travaille, par mesure de sécurité, sur la base d'une mémoire tampon. Dès l'appel de **vi** , on se trouve dans le mode commande. L'option **-r** permet la récupération du fichier avant un crash.

Commandes de base

a	passage en mode insertion après le curseur
i	passage en mode insertion avant le curseur
ESC	retour en mode commande

:	passage en mode ex
<rc>	début de la ligne suivante
ZZ	écrire le tampon et quitter vi
:w	écrire le tampon sauve sans sortir
:q	quitter vi si le tampon a été écrit
:q!	quitter vi sans sauvegarde
:sh	retour temporaire au shell , pour quitter le shell utiliser **Ctrl+D**.
:r!commande	exécution d'une **commande** Unix, le résultat est ajouté au fichier ouvert à la position du curseur

Déplacements dans la fenêtre

h ou backspace	à gauche d'un caractère
l ou espace	à droite d'un caractère
k	monter d'une ligne
j	descendre d'une ligne
+	début de la ligne suivante.
-	début de la ligne précédente.
H	place le curseur en haut de l'écran.
M	place le curseur au milieu de l'écran.
L	place le curseur en bas de l'écran.
w	début du mot suivant
b	début du mot précédent
^^	début de la ligne courante
$	fin de la ligne courante

Déplacements dans le fichier

^D	descend d'une demi-page.
^U	remonte d'une demi-page.
^F	descend d'une page.
^B	remonte d'une page.
nG	ligne numéro n.

Recherche de schémas

/ modèle	recherche le schéma en avant (par exemple [a-b]*) (après le curseur)
? modèle	recherche le schéma en arrière (avant le curseur)
n	recherche la prochaine occurrence du dernier schéma spécifié.

Recherche de caractères

f*car*	place le curseur où se trouve le caractère *car* dans la ligne courante, la recherche s'effectue vers l'avant.
F*car*	la recherche s'effectue vers l'arrière

Corrections

`rcar`	remplace le caractère sous le curseur par le caractère `car`.
`cw`	modifie le mot courant

Recherche et remplacement

```
:[Numligne1, Numligne2] s/Modèle/Remplacement/[gc]
```

`g` global sur tout le texte entre les deux lignes
`c` demande de confirmation

`:1,10 s/unix/Unix/g`	remplace **unix** par **Unix** de la ligne 1 à la ligne 10
`:1,$ s/unix/Unix/`	remplace la première occurrence d'**unix** par **Unix** de la ligne 1 à la fin du texte
`:.,$ s/unix/Unix/g`	remplace **unix** par **Unix** de la ligne active à la fin du texte
`:1,$ s/Mars/&upial/g`	remplace **Mars** par **Marsupial** de la ligne 1 à la fin du texte

Destructions

- En mode insertion :

`backspace`	efface un caractère

- En mode commande :

`x`	destruction du caractère sur le curseur
`(n)dd`	destruction de **n** lignes
`dw`	destruction du mot actif.
`db`	destruction du mot précédent.
`dH`	destruction de toutes les lignes entre la ligne courante et la première ligne de l'écran.
`dM`	destruction de toutes les lignes entre la ligne courante et la ligne du milieu de l'écran.
`dL`	destruction de toutes les lignes entre la ligne courante et la dernière ligne de l'écran.
`D`	efface jusqu'à la fin de la ligne

Autres commandes

`^G`	affiche le nom du fichier et le nombre de lignes
`u`	annule la dernière modification
`.`	répète la dernière commande

UNIX
Gestion de processus

5.1 Le concept de processus

Processus :

Un processus se compose de deux parties :
- un programme
- un environnement.

Le programme doit être chargé en mémoire et en cours d'exécution.

L'environnement est l'ensemble des informations fournies par le système d'exploitation au programme pour lui permettre une exécution correcte. Ces informations sont :
- numéro d'identification du processus (`PID`)
- numéro d'identification du processus parent (`PPID`)
- numéro d'utilisateur (`UID`)
- numéro de groupe (`GID`)
- durée de traitement utilisée (temps CPU) et priorité du processus.
- références du répertoire de travail actif (ex : Home)
- table de référence des fichiers ouverts (ex : tty01)

Création de processus

Un processus est crée par copie du processus créateur, dit père (mécanisme de fourche). Le père peut attendre la terminaison d'un fils. Le fils hérite de l'environnement du père (`TERM`, `HOME`,..., etc).

Le processus de démarrage du système (**pseudo-processus**) n'a pas de parent. Son PID est égal à 0.

Le processus `init` est le processus fils du pseudo-processus du démarrage. Son PID est égal à 1. Il est responsable, directement ou indirectement, de tous les processus en cours dans le système. Il fournit la base permettant d'afficher à l'écran le login et d'utiliser le shell.

Un utilisateur connecté signifie un processus shell est en cours.

Un processus « **deamon** » (ex : **swapp**...) est un processus en cours mais en arrière plan.

Un processus « **zombie** » est un processus qui persiste toujours parce qu'il attend un événement qui n'arrivera jamais, cela peut survenir à cause d'une erreur matérielle ou logicielle.

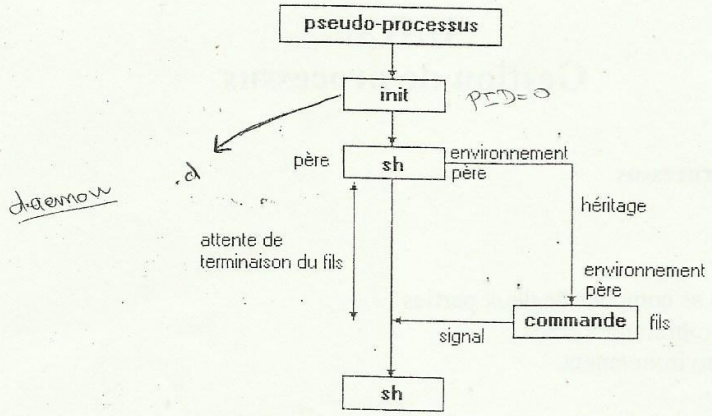


Fig. 5.1 Mécanisme de création de processus

Exemple :

```
$ ps
PID    TTY      STAT      TIME      COMMAND
6087   tty01     S        0:00      sh
6091   tty01     R        0:00      ps
```

Commentaires:

- deux processus : **sh** et **ps**
- **sh** a crée un fils
- **ps** est donc en cours (**Runing**)
- **sh** est suspendu (**Suspended**)
- l'environnement des deux processus est identique.(ex: même `tty`, même `usr`...)
- le processus fils **ps** signale sa terminaison à **sh** et meurt
- le processus **sh** redevient actif et renvoie le prompt **$**

Propriétés des processus :

- l'exécution des processus activables peut se faire en peudo-parallèlisme (temps partagé)
- plusieurs processus peuvent coopérer avec d'autres processus.
- un processus peut demander le remplacement de son code axecutable par le code d'au autre programme

5.2 Appels système pour la création de processus

L'appel système **fork**

L'appel **fork** permet la création d'un processus fils. Il retourne la valeur du PID du processus fils au processus parent ou une valeur négative en cas d'erreur. Il retourne la valeur 0 au processus fils.

```
#include <sys/types.h>
int fork(void)
```

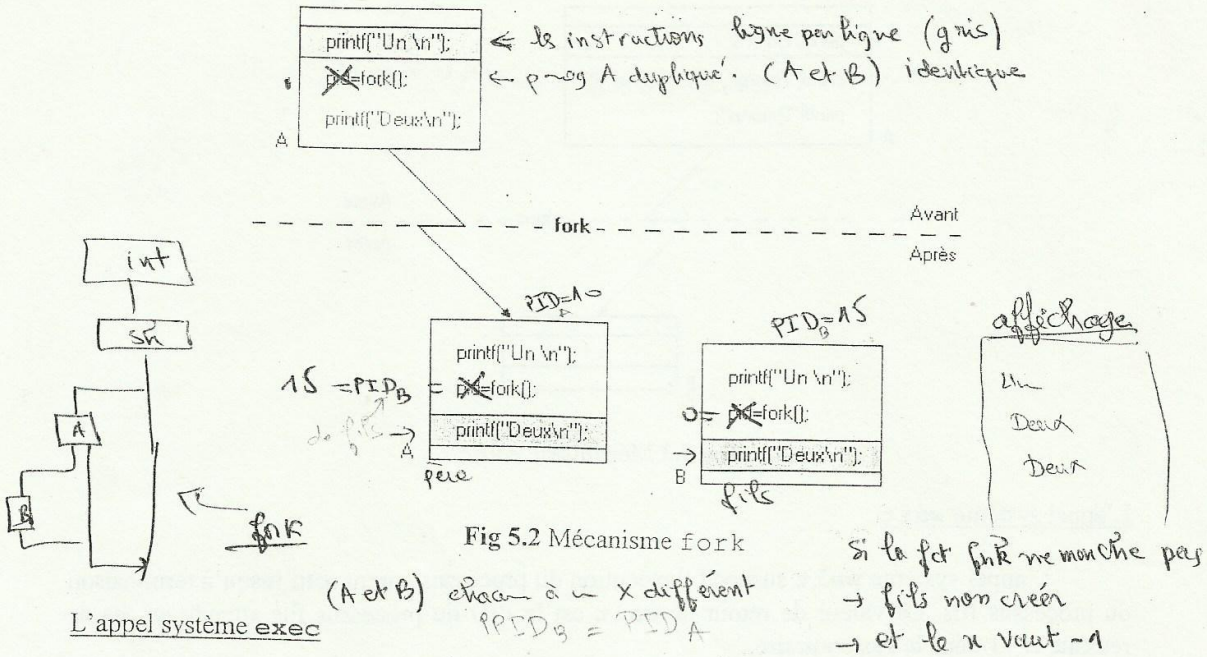


Fig 5.2 Mécanisme `fork`

L'appel système `exec`

L'appel système **`exec`** remplace le segment de code et de données d'un processus actif par un autre programme spécifié. Il renvoie une valeur de retour –1 en cas d'erreur et aucune valeur importante en cas de réussite.

```
#include <sys/types.h>
int execl(char *path, char *arg0, char *arg1,…,char *argn,
          (char*)0) ;
int execv(char *path, char *arg[]) ;
```

`path` : chemin du programme appelé pour être exécuté
`arg0` ou `argv[0]` : nom du programme appelé
`arg1,…,argn` ou `arg[1],…,arg[n]` : arguments du programme appelé
`(char*)0` : pointeur NULL pour marquer la fin de la liste des arguments.

Il existe d'autres versions de **`exec`** : **`execlp()`**, **`execvp()`**, **`execle()`** et **`execve()`**.

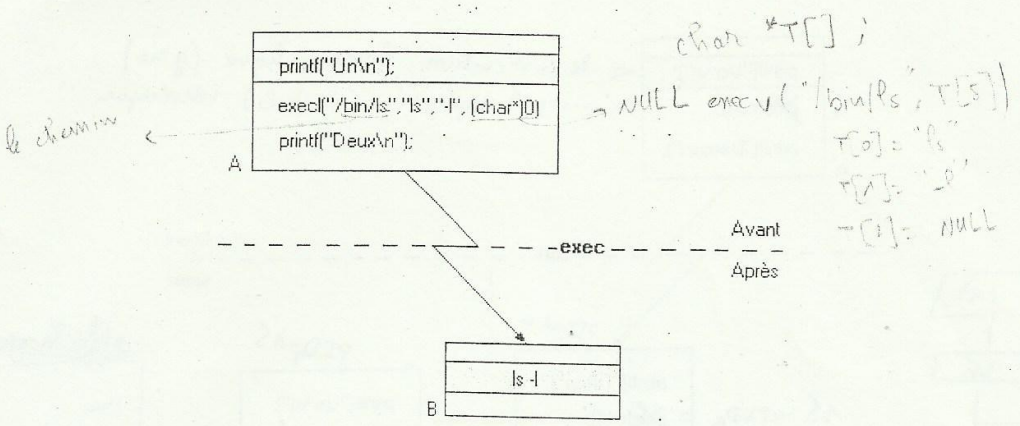


Fig 5.3 Mécanisme `exec`

L'appel système `wait`

L'appel système **`wait`** suspend l'exécution du processus parent actif jusqu'à terminaison du processus fils. La valeur de retour de **`wait`** est le PID du processus fils attendu en cas de réussite et –1 dans le cas contraire.

```
#include <sys/types.h>
int wait (int *status) ;
```

`status` est un argument dont lequel sont stockés un ensemble d'information concernant la terminaison du fils .

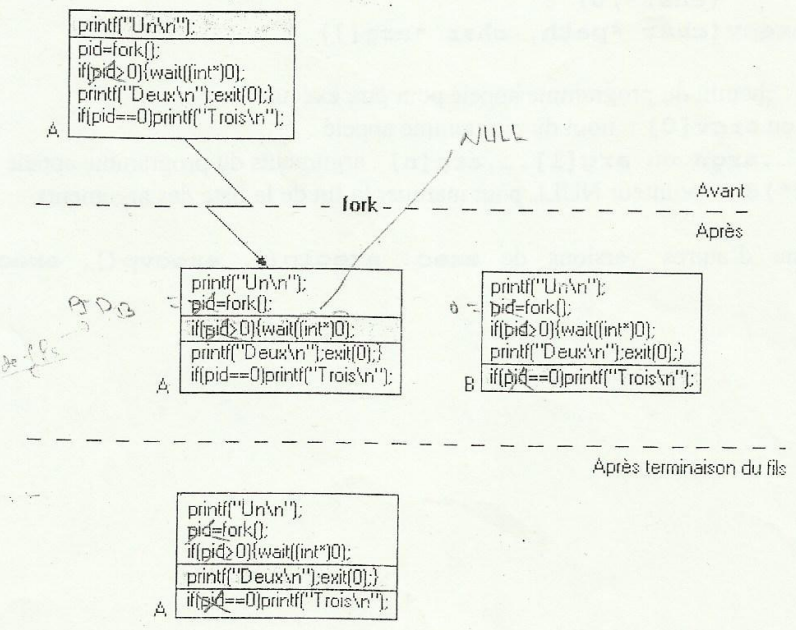


Fig 5.4 Mécanisme `fork` et `wait`

L'appel système `exit`

L'appel système `exit` est utilisé pour finir un processus. Il ne retourne pas de valeur et effectue les actions suivantes lors de son appel :

- fermeture des fichiers ouverts
- libération de la zone mémoire occupée par le processus
- envoie des informations au processus parent
- envoie de signaux à tous les processus fils

```
#include <stdlib.h>
void exit(status) ;
```

`exit` n'a pas de valeur de retour. `status` est une valeur retourné au parent (généralement un shell), pour le Bourne shell si cette valeur est 0 elle indique la réussite de `exit`.

L'appel système `kill`

L'appel système `kill` envoie un signal à un processus quelconque. La valeur de retour est –1 en cas d'erreur et 0 dans le cas de réussite.

```
#include <sys/types.h>
int kill(int pid, int signalnumber) ;
```

`pid` est celui du processus auquel sera envoyé le signal de numéro `signalnumber`.

Si `pid=0` le signal sera envoyé à tous les processus qui ont le même numéro de groupe que le processus émetteur.

Si **signalnumber=0** , le système vérifie seulement si le processus avec le PID indiqué existe.

Signal	Utilisation
1 (SIGHUP)	Envoyé par le processus parent à tous ses fils lorsqu'il termine son activité
2 (SIGINT)	Signal d'interruption d'un processus (Suppr ou Ctrl+C)
3 (SIGQUIT)	Equivalent à 2 sauf qu'il stocke dans le répertoire actif un état de la mémoire
9 (SIGKILL)	Le signal 9 ne peut être ignoré par aucun processus et sert à tuer
15 (SIGTERM)	Signal par défaut de `kill` pour mettre fin (terminer) aux processus

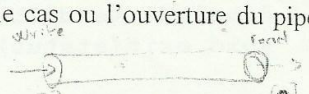
Table 5.1 Signaux standards

Communication entre processus

Les processus communiquent entre eux à l'aide de signaux ou à travers des pipes.

```
#include <stdio.h>
int pipe (int filedesc[2]) ;
```

L'appel système **pipe** utilise deux descripteurs de fichiers : **filedesc[0]** et **filedesc[1]**. Ces descripteurs sont utilisés pour distinguer les deux bouts du pipe dans lesquels on peut lire ou écrire. La valeur de retour est 0 dans le cas ou l'ouverture du pipe est fructueuse et –1 dans le cas contraire.



Exemple :

```
/Programme d'écriture/lecture d'un pipe*/
# include <stdio.h>
#define MSGTAILLE 16
main()
{ char mot[MSGTAILLE] ; int p[2], i ;
  char *message1= ''bonjour papa '' ;
  char *message2= ''bonjour maman'' ;
/*ouverture du pipe*/
  if(pipe(p)<0){printf(''Erreur d'ouverture du pipe '' ;
  exit(1) ;}
/*écriture dans le pipe*/
  write(p[1],message1, MSGTAILLE) ;
  write(p[1],message2,MSGTAILLE) ;
/*lecture du pipe*/
  for(i=0 ;i<2 ;i++){read(p[0],mot,MSGTAILLE) ;
  printf(''%s\n'',mot) ;}
  exit(0) ;
 }
```

5.3 Gestion des processus

Traitement en tâche de fond :

Le shell permet de lancer des processus sans attendre la fin de leur execution. Il suffit pour cela d'ajouter le **&** à la fin de la commande. Le shell est son processus enfant fonctionnent en parallèle, sans aucune coordination entre les deux, c'est une exécution asynchrone. Mais avant de revenir au prompt, apparaît le numéro du processus lancé.

Le traitement en tâche de fond ne peut se faire sans contraintes :

- le processus en tâche de fond ne doit pas attendre de saisie du clavier.
- le processus en tâche de fond ne doit pas retourner ses résultats à l'écran.
- si on quitte le shell par Exit ou Ctrl+D , on force également l'arrêt des processus en tâche de fond.

Exemple :

```
$ who > fichier &
4220
$
```

Commandes de gestion de processus

$ps [option] #affiche les informations concernant l'état des processus en cours

$wait [numéro du processus]

demande au shell d'attendre le processus en tâche de fond dans le PID est indiqué dans la commande. Si le numéro du processus est omis, le shell attend la fin de tous les processus. la commande **wait** peut être interrompue avec Ctrl+D.

$kill [-numéro du signal] PID [PID...]

envoie un signal aux processus en tâche de fond dont les PID sont indiqués dans la commande. Sans numéro du signal c'est le signal numéro 15(SIGTERM) qui sera automatiquement envoyé au processus destinataire.

$nohup commande [arguments]

La commande lancé avec **nohup** (no hang up) ignore l'arrivée du signal SIGHUP (signal envoyé par le père à ces fils). Elle ne sera pas tué au logout. Si l'utilisateur n'entreprend pas lui même une redirection de la sortie, toutes les sorties seront redirigées dans un fichier appelé **nohup.out**.

$nice -valeur commande [arguments]

La commande **nice** permet de lancer un processus tout en modifiant son facteur nice **NI**, c.à.d : sa priorité de traitement. Plus le facteur nice est grand plus le traitement est lent et vice-versa. La valeur normale de **NI** est généralement 20 (ou 0). Si on abaisse de 10 la valeur de **NI** sera 30 (`$nice -10 commande`). Si on augmente de 5 la valeur de **NI** sera 15 (`$nice --5 commande`).

$time commande [Argument(s)]

Elle affiche:
- le temps système utilisé (system) ((CPU) appels système)
- le temps utilisateur utilise (user) ((CPU) ordres du programme)
- le temps total, entre le début et la fin de la commande (réel)

$sleep <nombre>

sleep interrompe l'exécution d'une commande pendant un laps de temps. Le paramètre **nombre** désigne le nombre de secondes pendant lesquelles le processus actif est endormi.

Exemple :

```
$(Sleep 120; who > F) &
4536
$
# le fichier F ne sera crée qu'après 2 minutes
```

```
$at  heure [date] [+ increment]
> commande
> prog1
> prog2
  .
 Crtl+D
$
```

at permet de lancer une liste de commandes et de programmes à une date ultérieure.

Exemples :

$at 1600 <commande> # la commande sera lancé à 16^h 00mn.

$at 1715 fri <commande> # la commande sera lancé à le Vendredi 17^h15mn

$at noon <commande> # la commande sera lancé à midi pile.

5.3 Droits d'accès étendus pour les processus.

Notion de bits s

$chmod u+s programme ----> -rwsr-xr-x

Lorsque le **SUID-bit** (Set _User_ID-bit) est positionné pour un programme exécutable , tout utilisateur de ce programme possède les droits du propriétaire pendant l'exécution.

$chmod g+s programme ----> -rwxr-sr-x

Lorsque le **SGID-bit** (Set _Group_ID_bit) est positionné pour un programme exécutable, tout utilisateur de ce programme possède les droits du groupe propriétaire pendant l'exécution.

Exemple :

Les utilisateurs sont enregistrés dans le fichier système: `/etc/passwd` (ou `shadow`)

```
-rw-r--r--    root  sys    /etc/passwd
```

Tout utilisateur à le droit de lire `/etc/passwd` mais ne peut modifier directement son mot de passe (à l'aide d'un éditeur par exemple); une fonction permet d'effectuer la modification: `/bin/passwd`.

Si les droits d'accès de cette commande sont :

```
-rwxr-xr-x    root  sys    /bin/passwd
```

l'exécution de **/bin/passwd** s'effectue dans le contexte d'un utilisateur et à son niveau de privilège. Alors l'utilisateur qui n'est pas l'administrateur ne peut changer son mot de passe. Mais si on positionne le **SUID-bit**, il peut modifier son mot de passe.

```
-rwsr-xr-x    root sys  /bin/passwd
```

Remarques:

Le SUID-bit et le SGID-bit ne sont utilisables qu'avec des programmes binaires et non pas avec les scripts du Shell. Ils ne peuvent être attribués que par le propriétaire ou l'administrateur.

UNIX
Le shell

6.1 Notions et mécanismes de bases

le shell :

Le shell est l'interface de commande du système. Il peut travailler en deux modes:

- mode interactif
- mode programmé (**scripts**)

Fonctionnement du shell

Lors du lancement d'une commande, le shell effectue les opérations suivantes :
- analyse de la ligne de commande (attention aux caractères spéciaux)
- recherche du fichier contenant le programme de la commande.
- création du processus qui va executer ce programme
- chargement du programme et passage des arguments.

Les types de shell

Bourne Shell : sh
C shell : csh
Tcshell : tcsh
Kornshell : ksh
Bourne again shell : bash
.
etc.

La ligne de commande:

- La ligne de commande va du prompt du shell, jusqu'a <CR>
- Le premier mot est le nom d'un fichier executable, ou d'une commande interne au shell
- La commande peut être suivie par des options et/ou des arguments (séparés par des espaces)
- La ligne de commande n'est pas exécutée avant <CR>
- Etat de sortie (exist status) d'une commande :
  - * Terminaison correcte => exit 0
  - * Terminaison incorrecte => exit n° ≠ 0
- Le shell connait 4 types de lignes de commandes :
  - * Simple
  - * Séquentielle (chaînées)
  - * pipeline
  - * groupée.

simple :

- mode foreground : `$cmd`
- mode background : `$cmd &`

pipeline `$cmd1 | cmd2 | cmd3`

séquentielle `$cmd1; cmd2; cmd3`

groupées `$(cmd1; cmd2; cmd3)` ou `${cmd1; cmd2; cmd3}`
avec lancement d'un nouveau shell

Redirections

`<`	redirection de l'entrée standard
`>`	redirection de la sortie standard (création)
`>>`	redirection de la sortie standard (ajout)
`2>`	redirection de la sortie d'erreurs (création)
`2>>`	redirection de la sortie d'erreurs (ajout)
`<<`	redirection spéciale de l'entrée standard (Herescripts)
`2>&1`	redirection comme de la sortie standard et de la sortie d'erreurs vers la sortie standard
`1>&2`	redirection de la sortie standard vers la sortie d'erreurs.

Génération de noms (Jokers): `?` `*` `[ ]`

6.2 Les variables shell

Les variables:

Deux types de variables
- crées et maintenues par le système
- crées par l'utilisateur.

La valeur d'une variable est une chaîne de caractères **`nom_variable =`** `chaîne`

Exemple:

```
cours = Unix          ---> chaîne 'Unix'
poids = 156           ---> chaîne '156'
```

L'interprétation numérique d'une chaîne de chiffres se fait à l'aide des commandes **`test`** et **`expr`** .

Remarque : chaîne vide `x=` , `x=''`, `x=""`.

L'accès à la valeur d'une variable se fait avec la commande **`echo`**

Exemple

```
$echo cours
 cours
$echo $cours
Unix
$
$echo la valeur de cours est $cours
la valeur de cours est Unix
$
```

Les variables shell pré-définies

Ces variables sont appelées variables d'environnement. La commande **set** permet d'imprimer ces variables.

```
$set
EXINIT='set ai nu'                  (*instruction d'initialisation de l'éditeur*)
HOME=/USR/USERS/Ahmed               (*répertoire courant*)
IFS=                                (*séparateur des mots dans une commande
                                     par défaut l'espace*)
LOGNAME=Ahmed                       (*nom de l'utilisateur*)
MAIL=/USR/mail/Ahmed                (*boite à lettre de l'utilisateur*)
PATH=.:/bin:/USR/bin                (*noms des répertoires dans lesquels le shell
                                     recherche successivement les commandes*)
PS1=$                               (*1er prompt*)
PS2=>                               (*2ème prompt lors de l'entrée d'une
                                     commande sur plusieurs lignes*)
TERM=VT100                          (*type de terminal utilisé*)
.....
```

set permet aussi de modifier les paramètres du shell

Remarque : les fonctionnalités de **set** dépendent du shell utilisé

Le fichier `.profile`

Ce fichier contient les instructions initiales de toute session . L'ouverture d'une session conduit automatiquement à l'exécution du `.profile`.

```
$cat .profile
HOME=/USR/USERS/Ahmed
MAIL=/USR/mail/Ahmed
PATH=.:/bin:/USR/bin
PS1=$
export HOME MAIL PATH TERM PS1
date
echo Salut Ahmed
$
```

L'ouverture d'une session avec ce fichier est la suivante :

```
Login : Ahmed
Password :**********
Mon Dec 28:02 11:15
Salut Ahmed
$
```

Les variables shell locales et globales

Une variable ordinaire est une variable locale; elle n'est connue que dans le shell où elle a été créée. Une variable de même nom créée dans un sous-shell sera une variable distincte. La valeur de la même variable dans le shell père n'est pas modifiée. Ainsi, tout shell a ses propres variables locales.

```
$cours = Unix
$echo $cours
 Unix
$sh------>$echo $cours
          # variable non définie dans le sous shell
          $cours=DOS
          $echo $cours
           DOS
          $Ctrl+D          # retour au shell initial
$echo $cours
 Unix
$
```

La commande **export** permet de transformer une variable locale en une variable globale, c.à.d : tous les sous-shell possèdent une copie de cette variable.

```
$cours = Unix
$export $cours
 Unix
$sh------>$echo $cours
           Unix            # variable connue dans le sous shell (héritage)
          $cours=DOS
          $echo $cours
           DOS
          $Ctrl+D          # retour au shell initial
$echo $cours
 Unix
$
```

Remarque :

Les variables peuvent être exportées vers les sous-shell, mais ne peuvent pas être renvoyées vers le shell initial.

6.3 Les shell scripts

Script :

Un shell script est un fichier basé sur des commandes shell (**read** et **echo**) ainsi que sur l'usage des variables.

Exemple:

```
SC :
echo Cher utilisateur quel est ton nom \?
read nom
echo Ravi de te connaître $nom
```

Résultat

```
$sh SC
Cher utilisateur quel est ton nom ?
Ahmed
Ravi de te connaître Ahmed
$
```

Remarque :

- On peut lire une liste de variables `$read nom prenom`
- Il faut utiliser **`chmod u+x`** pour rendre un script exécutable

Arguments

Dans les scripts les arguments sont référencés par leur position dans la liste des arguments :

$1 premier argument
$2 deuxième argument
.
.
etc.

set permet aussi de remplir les paramètres positionnels

Exemple :

```
$set Dos  Unix Vms
$echo $1
Dos
$echo $2
Unix
```

Conventions :

$0 nom de la commande elle-même
$* liste des arguments
$# nombre d'arguments

Exemple : Script "Expliquer"

```
echo Je suis le shell script $0
echo Mes arguments sont $*
echo Le premier argument est $1
echo Le nombre de mes arguments est $#

$Expliquer MOI LES ARGUMENTS
 Je suis le shell script Expliquer
 Mes arguments sont MOI LES ARGUMENTS
 Le premier argument est MOI
 Le nombre de mes arguments est 3
$
```

Remarque:

Les pipes et les redirections peuvent être utilisées à l'interieur et à l'exterieur des SC

Substitution de commande :

Lorsqu'une chaîne de caractères est entourée par ` `, elle est interprétée comme une commande et remplacée par son résultat.

Exemple

```
$echo date
date
$echo `date`
Mon dec 29 15:40 2002
$
```

Les caractères spéciaux

- Expressions régulières : ? * [ ]
- Séparateurs : ' " { } ; , ( )
- Méta-caractères : \ $!
- Opérateurs particuliers : | &

Mécanismes d'échappement :

\	négation d'un méta-caractère.
'...'	(simples quotes) négation de tous les méta-caractères
"..."	(doubles quotes) interprétation de $ ` \

Exemple

```
$echo `date`
 Mon Aug 18 16:05  2002
$echo 'Nous sommes le `date`'
 Nous sommes le `date`
$echo "Nous sommes le `date`"
 Nous sommes le Mon Aug 18 16:10 2002

$echo 'Ceci est une <cr>
>très longue <cr>
>phrase' <cr>
Ceci est une
très longue
phrase
$
$echo voici une étoile : \*
 voici une étoile : *
$
```

Les commentaires

Les commentaires dans un script doivent toujours commencer par le caractère #

Variables spéciales

$#	nombre de paramètres positionnels
$-	options du shell
$?	état de sortie de la dernière commande exécutée.
$$	n° de processus du shell courant
$!	n° du dernier processus en background
$0	nom de la commande en cours
$*	liste des paramètres positionnels

Exemple :

```
$date >> F
$echo $?
0                (*état de la dernière commande*)
$date >> F &
4321             (*PID du processus en background*)
$echo $!
4321       (*PID du dernier processus en background*)
$
```

6.4 Programmation shell

Boucle **for** :

```
for variable in valeur1 valeur2 ...
do
commandes
done
```

Exemples :

SC1 :

```
for i in UNIX DOS VMS
do
echo je connais quelqu\'un qui a suivi un cours $i
done
```

```
$SC1
Je connais quelqu'un qui a suivi un cours UNIX
Je connais quelqu'un qui a suivi un cours DOS
Je connais quelqu'un qui a suivi un cours VMS
```

```
$for i in 1 2 3 ; do echo $i ; done
```

```
$for i in 1 2 3 <cr>
> do <cr>
> echo $i <cr>
> done <cr>
```

Exemple de for avec passage d'arguments

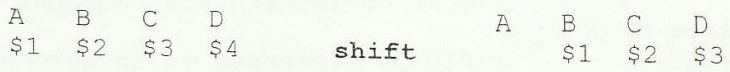
SC :

```
for i in $*     #passage d'arguments
do
echo je connais quelqu\'un \
qui a suivi un cours $i
done
```

```
$SC UNIX DOS VMS     #résultat identique à SC1.
```

Exemple de for avec shift

shift effectue un décalage à gauche des paramètres positionnels



SC :

```
message = $1
shift
for personne in $*
do
mail $personne < $message
done
#envoie d'un message à plusieurs utilisateurs

$SC M  Ahmed Aziz Fatima
```

Exemple de for et génération de noms

```
for F in *.c
do
cp $F  /home/ieea4/Exercices
done
#sauvegarde de fichiers source .c
```

Exemple de for et interactivité

SC :

```
echo sujets\?
read sujets
for i in $sujets
do
echo Je connais quelqu \'un\
qui a suivi des cours  $i
done

$SC
sujets ?
UNIX DOS VMS
Je connais quelqu'un qui a suivi un cours UNIX
Je connais quelqu'un qui a suivi un cours DOS
Je connais quelqu'un qui a suivi un cours VMS
```

Exemple de for et substitution de commande

SC :

```
for i in `cat F`     # F contient UNIX DOS VMS
do
echo Je connais quelqu \'un\
qui a suivi des cours $i
done
```

Boucle **while** :

```
while  commande_de_controle
do
commandes
done
```

A chaque pas de boucle la commande_de_contrôle est exécutée :
- si elle retourne au shell la valeur 0 (**exit**(0)) ==> exécution du corps de la boucle
- sinon ==> sortie de la boucle.

Ainsi, la boucle **while** s'exécute jusqu'à ce que la commande_de_contrôle ait un état de sortie ≠ 0 (arrêt sur erreur de la commande_de_contrôle)

Exemple :

SC :

```
# recherche du 1er fichier ne contenant pas un mot
# usage : SC   mot   liste de fichiers
mot = $1 ; shift
while grep $mot $1 > F  #résultat dans F
do
shift
done
echo $1 premier fichier ne contenant pas $mot
```

Boucle **until**

```
until  commande_de_controle
do
commandes
done
```

La boucle **until** s'exécute tant que la commande_de_contrôle a un état de sortie ≠ 0 (arrêt sur succès de la commande_de_contrôle)

Exemple :

SC :

```
# recherche du premier fichier contenant un mot
mot = $1 ; shift
until grep $mot $1 > F  #résultat dans F
do
shift
done
echo $1 premier fichier contenant $mot
```

L'instruction **case**

```
case valeur in
     choix1) commandes;;
     choix2) commandes;;
       ...
          *) commandes;;      # cas par défaut
esac
```

Exemples :

SC1 :

```
echo 'n° du programme que vous souhaitez? '
echo '1-date        2-who'
echo '3-ls          4-pwd'
read choix                  # entrée de la valeur
case $choix in
     1) date;;
     2) who;;
     3) ls;;
     4) pwd;;
     *) echo 'choix incorrect, désolé!';;
esac
```

SC2 :

```
echo Quel est le meilleur ami de l\'homme\?
echo Tigre Lion Chien Serpent
read reponse
case $reponse in
     Chien)echo Réponse correcte;;
     Tigre|Lion|Serpent)echo Réponse incorrecte;;
     *)echo $reponse, réponse nom proposée;;
esac
#le symbole | n'est pas ici le symbole de pipe,
#mais le ou logique
```

SC3 :

```
for F in $*
do
case $F in
     *.old) rm $F;;
       *.c) mv $F  /home/ieea/Exercices ;;

esac
done
```

Remarque :

- Les labels peuvent utiliser: ? [ ] *
- Les commandes peuvent utiliser les pipes et les redirections.

L'instruction `if`

```
if commande_de_contrôle
then
     commandes
else
     commandes
fi
```

`else` est optionnel

Exemples :

SC :

```
# destruction d'un fichier parmi deux identiques
# usage : SC1   F1   F2
if cmp -s $1 $2          # -s : état de sortie de cmp
then
     rm $1; echo $1 et $2 sont identiques
else
     echo $1 et $2 sont des fichiers distincts
fi
```

On peut imbriquer des tests if de la manière suivante :

```
if commande_de_contrôle_1
then
     commandes_A
elif commande_de_contrôle_2
then
     commandes_B
else
     commandes_C
fi
```

Si succès (exit 0) de commande_de_contrôle_1 alors
exécution des commandes_A
sinon
si succès (exit 0) de commande_de_contrôle_2 alors
exécution des commandes_B
sinon
exécution des commandes_C

L'instruction **test**

La commande **test** permet d'effectuer des tests d'existence et de comparaison.

Exemple :

```
$nom = Ahmed
$test $nom
$echo $?          #état de sortie de la dernière commande
0                 #existence de la variable nom
$test $prenom
$echo $?
1                 #inexistence de la variable prenom
$
```

Remarques :

test permet d'effectuer divers types de tests :
* test d'état de fichier
* test de comparaison de chaînes
* test de comparaison numérique

Exemples :

SC1 :

```
#deviner un nombre caché entre 1 et 50
echo Je pense a un nombre entre 1 et 50
echo Lequel \?
read reponse
until test $reponse=33
do
echo 'dommage, essayez encore!'
read reponse
done
echo 'Bien vu!'
```

```
$SC1
Je pense a un nombre entre 1 et 50
Lequel?
5
Dommage, essayez encore!'
33
Bien vu!
$
```

SC2 :

```
# deviner le meilleur OS
if test $1=Unix      # $1 premier paramètre
then
     echo je suis d\'accord
else
     echo Je n\'ai jamais entendu parler de $1
fi
```

Test d'état de fichier

```
test -r F          #vrai, si F existe avec le droit en lecture
     -w F          #vrai, si F existe avec le droit en ecriture
     -f F          #vrai, si F existe et n'est pas un répertoire
     -d F          #vrai, si F existe et c'est un répertoire
     -s F          #vrai, si F existe et sa taille > 0
     -x F          #vrai, si F existe et c'est un exécutable
```

le contraire : ! -r F

Exemple :

```
$test -w .profile; echo $?
0                                   # vrai
$test -w /etc/passwd ;echo $?
1                                   # faux
$
```

Test de comparaison de chaînes

```
test chaîne_1 = chaîne_2        # égalité
test chaîne_1 != chaîne_2       # non égalité
test -n chaîne                  # longueur non égal à 0
test -z chaîne                  # longueur 0
test chaîne                     # chaîne inexistente ou vide
```

Test de comparaison numérique

```
test n1 -eq n2      # égalité
     n1 -ne n2      # non égalité
     n1 -gt n2      # supérieur
     n1 -lt n2      # inférieur
     n1 -ge n2      # supérieur ou égal
     n1 -le n2      # inférieur ou égal
```

Exemples :

```
test 5 = 5          # vrai comparaison de chaînes
test 5 = 05         # faux comparaison de chaînes
test 5 -eq 05       # vrai comparaison numérique.
```

test et opérations logiques

```
!           #négation
-a          #et logique
-o          #ou logique
```

Exemples :

```
test !-r F
#vrai, si F existe et si pas de droit en lecture

test -r F1 -a -w F2
# vrai, si droit en lecture sur le fichier F1
# et si droit en écriture sur le fichier F2

test -r F1 -o -w F1
# vrai, si droit en lecture ou en écriture sur F1
```

formes réduites :

[...] équivalent à **test**

```
[ $nom = Ahmed ]        équivalent à    test $nom = Ahmed
[ -f fichier ]          équivalent à    test -f fichier

[ $age -gt 9 -a $age -le 20 ]
équivalent à  test $age -gt 9 -a $age -le 20
```

Remarque : [$nom = Ahmed] => erreur
[$nom = Ahmed] => correct

Calcul arithmétique par **expr**

expr interprète et évalue une expression arithmétique; le résultat est produit en sortie.

Exemple

il faut travailler avec des "entiers" !!

```
$expr 3+5
8
$expr 9-2
7
$expr 4\*3
12
$expr 12/3
4
```

SC :

```
# conversion minutes-secondes
case $# in
     0) echo pas d\'argument\!; exit 1;;
     *) min = $1;;
esac
while test $1
do
echo `expr 60 \* $min`
shift; min = $1
done
```

nbre d'arguments ; parcourt les commandes ; termine la commande.

$ annule l'effet de *

Factorielle :

```
#calcul récursif
case $1 in
1)echo 1;;
*) y=`Factorielle `expr $1 - 1\``;echo `expr $1 \* $y`;;
esac
```

$n! = n \cdot (n-1)!$

$f(n) = n \cdot f(n-1)$

fact(n-1)

$ ./sc 4

4 fact(3)
3 - fact(2)
2 - fact(1)
1

Fonctions prédéfinies

break pour sortir d'une boucle

Exemple :

+ return → sortie de fct → continue passe à l'instr suivant

SC :

```
while True
do
echo -e "saisie: \c"     # \c pour non retour à la ligne
read a
if [ -z $a ]
then
break
fi
.....
done
```

```
fn (i=0 ; i<10 ; i++)
{ if (T[i]<0) continue
else
{ S= S+T[i]; }
printf ("%d", S);
}
```

continue pour passer à l'itération suivante

Exemple :

SC :

```
echo "saisir le texte (q=quit, 'n=continue)"
while True
do
echo -e "nom: \c"
read nom
case $nom in
     q) break;;
     n) continue;;
esac
......
done
```

eval réitère l'exploitation d'une ligne du shell

Exemple :

SC :

```
$a=Unix
$b=a
$echo $b
a
$
$ eval    "echo \$$b"
Unix
$
```