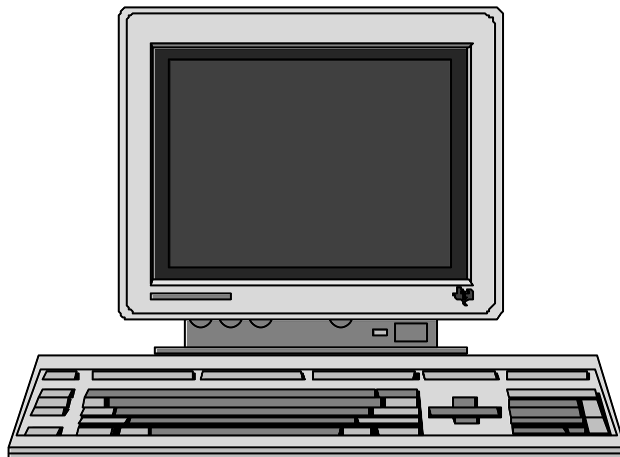
UNIVERSITE de CERGY

UNIX System V
Programmation en langage C



José GILLES - 1995

IUP GENIE ELECTRIQUE

I) Approche du système UNIX

1) Historique
2) Introduction
3) Architecture
4) La concurrence
5) Connexion et déconnexion
    a) Connexion
    b) Changement de mot de passe
    c) Qui est connecté ?
3) Le système de fichiers
    a) Architecture
    b) Droits d'accès
    c) Les répertoires
    d) Les fichiers réguliers
    e) Les liens
    f) Modifier les droits d'accès, le propriétaire, le groupe
    g) Recherche de fichiers
    h) Les répertoires principaux du système
4) Le shell et l'environnement
    a) Les commandes
    b) L'environnement
    c) Le script de configuration .profile ou .login
    d) La redirection des entrées / sorties
5) L'éditeur vi
6) Le gestionnaire d'impression lp
    a) Requêtes d'impression
    b) Suivi des requêtes
7) Gestion des processus
    a) Généalogie
    b) Environnement et héritage d'un processus
    c) Processus en tâche de fond
    d) Suspension du processus en cours
    e) Orphelins, démons et zombies
8) Programmation de scripts shell
    a) Programmation de sh et ksh
    b) Les spécificités du C-shell
9) Sauvegarde des données - Rapports avec MS-DOS
    a) tar
    b) Les rapports avec MS-DOS
10) Les terminaux texte
11) Les réseaux
    a) Un aperçu de TCP/IP
    b) Quelques services offerts par TCP/IP
12) Le e-mail

a) L'expédition de e-mail
b) La consultation

II) Fonctions de manipulation de fichiers et répertoires

1) Opérations sur les fichiers
a) Les fonctions de bas niveau
b) Les fonctions de haut niveau
2) Les fonctions de manipulation de fichiers
3) Les fonctions de manipulation de répertoires
a) Opérations sur les répertoires
b) Recherche d'un fichier

III) Gestion des processus

1) L'appel à exec...( )
2) L'appel à fork( )
3) Terminaison d'un processus
4) L'appel à wait...( )
5) Récupération des paramètres
6) Fonctions de récupération de pids
7) Manipulation de l'environnement
8) L'appel à system( )
9) Redirection des entrées / sorties

IV) Les signaux

1) Le principe des signaux
2) Les signaux du noyau
3) Les signaux du terminal
4) Les signaux logiciels
5) Primitives de gestion des signaux

V) Curses et terminfo

1) Le principe
2) Gestion de l'écran en mode curses
3) Initialisation du mode curses
4) Manipulations dans la fenêtre standard
5) Fenêtres et sous fenêtres

VI) Les pipes

1) Description
2) Les tubes locaux
3) Les tubes nommés

VII) Les sockets

1) Description
2) Les fonctions de manipulation de sockets

a) Pour un processus client
b) Pour un processus serveur
3) Les noms de machines et les adresses IP
4) Fonctions de conversion

VIII) Les outils de développement et mise au point

1) Le compilateur cc
2) Le gestionnaire de projets make
3) La mise au point

ENVIRONNEMENT UNIX
-
PROGRAMMATION EN C

IUP GE 3EME ANNEE

Matériel requis: Une station fonctionnant sous Unix
Environnement: UNIX SYSTEM V

I) Approche du système UNIX.

1) Historique:

Le système UNIX, bien implanté aujourd'hui dans les milieux universitaires
et dans les entreprises est né dans les années 70 dans les laboratoires d'AT&T
(American Telegraph & Telephone), BELL et Western Electric. Il s'agissait alors
de développer un système d'exploitation moderne (pour l'époque !).

Il faut, pour situer le contexte, rappeler qu'à l'époque l'informatique en était encore à l'aire
des méga-calculateurs qui ingurgitaient les programmes sous forme de cartes perforées
et retournaient leurs résultats sous forme de listings imprimés...

Dans ce contexte est apparue la nécessité d'établir un dialogue interactif entre
le programmeur et le calculateur, via un clavier d'où les données seraient saisies directement
—sans passer par l'intermédiaire de cartes perforées et d'opérateurs de saisie—
et via un écran—capable d'afficher une ligne de texte à la fois.
La notion de télétype était née.

Le système en question devait même pouvoir supporter plusieurs terminaux (télétypes)
ce qui sous entendait un système d'exploitation multiutilisateur donc multitâche.

Il fut décidé également d'introduire un nouveau système de fichiers hiérarchisé en répertoires.

Et enfin il fallait mettre l'accent sur l'interpréteur de commandes (le shell) et les utilitaires qui permettraient aux programmeurs de tirer le meilleur parti du système d'exploitation UNIX.

Les premiers travaux eurent lieu sur une plate-forme DEC (Digital Equipement Corporation)
autour de D. Richie, R. Canaday et B. Kernighan en 1969.

Dans un souci de portabilité il fut décidé que le nouveau système UNIX devait faire le moins possible appel à l'assembleur, très dépendant du matériel utilisé.

Kernighan et Richie décidèrent donc de développer l'essentiel de l'O.S. (Operating System) dans un langage évolué qui fut mis au point pour la circonstance, en 1972, à partir d'un langage existant, le langage B.
Depuis, le langage C est indissociable d'UNIX.
Seul le noyau (kernel) du système d'exploitation nécessite encore de l'assembleur.

Devenu portable, le système UNIX a intéressé de nouveaux constructeurs, parmi lesquels on comptait en 1975: DEC, Data General, Hewlett-Packard, ...

Avec la diffusion apparut aussi la diversification, chacun voulant apporter une touche spécifique à son système UNIX:

Data General diffusa sa version DG-UX
Hewlett-Packard diffusa sa version HP-UX
...

Les universités américaines se mirent aussi à perfectionner UNIX, et l'on doit beaucoup à l'université de Berkeley (Californie) qui a développé une version—hélas un peu différente des autres—connue sous le nom d'UNIX BSD
(Berkeley Software Developpement)

Dès lors UNIX s'est scindé en deux grandes familles, UNIX System III puis System V d'une part et UNIX BSD d'autre part.

Aujourd'hui UNIX est disponible sur de nombreuses plates-formes matérielles, avec quelques variations:

Citons entre autres:

Hewlett-Packard	HP-UX	System V
DEC	DEC-UX	System V
IBM (RS 6000)	AIX	System V
SUN	SunOS	BSD
	Solaris	System V
IBM (PC)	SCO UNIX	System V
	UnixWare	System V
	Interactive Unix	System V
	Solaris PC	System V
...		

Actuellement, les versions distribuées sont BSD 4.3 ou BSD 4.4 et System V Release 3.2 ou System V Release 4.

La multiplication des systèmes UNIX et leur divergence progressive ont entraîné des tentatives de normalisation; en effet, la portabilité du langage C d'un système à un

autre n'est plus ce qu'elle était: un programme C nécessite toujours des aménagements pour passer d'une plate-forme à une autre.

Les tentatives de normalisation des commandes, des utilitaires et du langage C sont venues de l'ANSI (American National Standard Institute) et de regroupements de constructeurs et de distributeurs de systèmes UNIX, parmi lesquels on peut citer POSIX en 1986, X/OPEN en 1988 et OSF/1 (Open Software Foundation).

2) Introduction au système UNIX:

Dans la suite nous nous consacrerons uniquement aux systèmes UNIX de type System V Release 3.2 ou Release 4.

Les systèmes UNIX offrent aujourd'hui bien plus de possibilités que leurs ancêtres.
Les télétypes ont été remplacés par des terminaux alphanumériques—encore nommés tty—puis par des terminaux graphiques.
Les périphériques accessibles sont très nombreux: lecteurs de disquettes, de CD-ROM, de DAT, streamers, scanners, imprimantes...

Les réseaux qui étaient initialement constitués de liaisons série ont été remplacés par des réseaux ethernet et le protocole TCP/IP est devenu un standard de fait dans l'environnement UNIX.

Tirant profit des réseaux et des terminaux graphiques, une interface graphique est venue compléter l'interface en mode texte typique d'UNIX. Cette interface est fondée sur X-Window, un système graphique client/serveur développé au MIT (Massachusetts Institute of Technology) qui en est actuellement aux versions X11 Release 5 ou X11 Release 6.
Disons que X-Window est à UNIX ce que Microsoft Windows est à MS-DOS.

UNIX est donc un système d'exploitation multitâche, multiutilisateur, extrêmement puissant.
Il existe des serveurs UNIX à la dimension d'un simple groupe de travail, d'une petite entreprise ou d'une grande société, gérant jusqu'à plusieurs centaines ou milliers de terminaux.

3) Architecture d'UNIX:

Le système UNIX est installé sur un serveur ou une station de travail.
Il comprend plusieurs couches, le **noyau** (kernel), qui est le coeur du système (écrit en partie en assembleur), les **drivers** de périphériques qui assurent l'indépendance du noyau vis à vis des périphériques et du matériel, les **shells** qui permettent le dialogue avec les utilisateurs, les **utilitaires** qui permettent de gérer et d'administrer le système, et les **applications** destinées aux utilisateurs.

Le diagramme suivant résume les liens entre ces différentes couches:

4) La concurrence:

Certes UNIX a fait son chemin, mais d'autres systèmes d'exploitation ont vu le jour depuis sa création et lui font parfois de l'ombre...

Citons par exemple:

Système	Distributeur	Plate forme
VMS	DEC	VAX
OS/2	IBM	PC
Windows NT	Microsoft	PC
Netware	Novell	PC

Les uns et les autres présentent des avantages et des inconvénients
que nous ne discuterons pas ici.

Vous remarquerez dans ce qui suit des similitudes entre UNIX et MS-DOS...
Il semble en effet que les concepteurs de CP/M et de MS-DOS dans les années 80 se soient inspiré d'UNIX, en conservant le minimum vital pour en faire un système d'exploitation monotâche et monoutilisateur qui puisse fonctionner sur un modeste PC.
(Mais cela est une autre histoire !)

5) Connexion et déconnexion:

Dans ce qui suit on suppose que le lecteur dispose d'un terminal UNIX en mode texte.
Nous n'aborderons pas l'environnement X-Window.

a) Connexion:

Chaque serveur ou station de travail UNIX est sous la responsabilité d'un administrateur qui se charge d'assurer le bon fonctionnement du système—il peut s'agir du principal utilisateur ou d'un informaticien spécialisé. Cet administrateur (superuser) dispose de tous les droits dans tous les domaines, tandis que les autres utilisateurs ont des droits limités. Par exemple, seul l'administrateur peut en général éteindre ou redémarrer le système.

Chaque utilisateur d'un système UNIX doit se faire connaître auprès de l'administrateur du système qui lui crée un compte utilisateur c'est à dire:

- Un répertoire de travail où il peut faire à peut près ce qu'il veut, nommé Home.
- Un nom d'utilisateur (login) qui identifie l'utilisateur auprès du système. (En général 8 lettres au plus)
- Un mot de passe (password) confidentiel qui permet d'authentifier le login. (En général 8 lettres ou chiffres au plus)

💣 Attention: Signalons de suite qu'UNIX est très pointilleux sur les minuscules et majuscules pour les login, les mots de passe et les commandes.

L'administrateur a pour login le nom root.
Son mot de passe est bien sûr le plus secret de tous !

Toute séance de travail sur un système UNIX commence par la phase d'identification login + mot de passe.
Le mot de passe tapé au clavier ne s'affiche jamais à l'écran pour des raisons de confidentialité.

Une fois la connexion réussie, l'utilisateur reçoit généralement un message de bienvenue, puis se retrouve face à l'interpréteur de commandes (shell) qui se manifeste par un prompt '$' ou '*' ou '%' ... en l'attente d'une commande.

Nous verrons plus loin que les commandes ne manquent pas !

Dans l'immédiat voyons comment mettre fin à une séance de travail:
il est impératif de se déconnecter, sans quoi un autre utilisateur pourrait travailler sous votre identité...

La déconnexion peut se faire de diverses manières, selon le système et le shell utilisé:

- En tapant logout ↵
- En tapant exit ↵
- En faisant ^D (CTRL_D) au clavier

Après déconnexion le système génère une nouvelle demande de login.

<u>Exemple:</u>

```
login: gilles ↵
passwd: -------- ↵

Friday September 15 1995, 9:00 am
Welcome on server u_cergy, gilles
$
.
.
.
$ logout ↵
Logout from server u_cergy.

login:
```

b) Changement de mot de passe:

Le mot de passe initial de chaque utilisateur est souvent attribué par l'administrateur root.

Sauf exception, l'utilisateur peut en changer régulièrement et à sa guise.
Certains systèmes imposent d'ailleurs un changement périodique de mot de passe;
le système demandera alors après connexion un nouveau mot de passe.

L'administrateur peut imposer aux mots de passe une complexité minimale:
tant de lettres et/ou de chiffres au minimum.

La commande passwd permet de changer de mot de passe:

Syntaxe: passwd
passwd *login_name*

Le système demande alors d'entrer deux fois le nouveau mot de passe pour l'utilisateur indiqué, ou par défaut pour l'utilisateur connecté.

Seul root peut en fait changer le mot de passe de quelqu'un d'autre !

c) Qui est connecté ?

Le système UNIX étant multitâche et multiutilisateur il y a de fortes chances pour que vous ne soyez pas le seul connecté au serveur. Des utilisateurs peuvent se connecter à partir de différents types de postes:

- La console: le poste de travail situé sur le serveur ou la station.
- Les terminaux série: terminaux texte reliés au serveur par une liaison série ou une ligne téléphonique et un modem.
- Les terminaux réseau: terminaux texte en réseau.
- Les terminaux X: terminaux graphiques sous X-Window.

Un terminal est en général un matériel spécifique doté d'un clavier, d'un écran et d'une interface de communication, sans mémoire propre, et qui ne saurait fonctionner sans le serveur.
Il peut s'agir aussi d'un PC ou d'un MAC doté d'une interface de communication série ou réseau et d'un logiciel de communication ou d'émulation de terminal.

Schéma d'un système UNIX:

De nombreuses commandes permettent de savoir qui est connecté au serveur, et où, car chaque terminal est identifié par un nom auprès du serveur.

La commande who fournit la liste des utilisateurs connectés, le terminal sur lequel chacun travaille et les jour et heure de connexion:

Syntaxe: who
who *am i*

La commande who *am i* permet à un utilisateur d'obtenir sa propre identité

(si, si, cela peut servir...) et son propre terminal.

Exemple:

$ who ↵			
root	tty01	Sep 15	9:15 am
gilles	ttyp0	Sep 15	10:20 am
dupont	ttyp1	Sep 15	10:07 am
$ who am i ↵			
gilles	ttyp0	Sep 15	10:20 am

Pour n'obtenir que le nom du terminal on peut se contenter de la commande tty

Syntaxe: tty

On obtient alors le nom complet du terminal rattaché au répertoire /dev

Exemple:

```
$ tty ↵
/dev/ttyp0
```

On peut par ailleurs consulter la date, l'heure, et même un calendrier perpétuel avec les commandes date et cal:

Syntaxe:
date
date mmddHHMM[yy]
cal [mm][yyyy]

La seconde forme de date est réservée à root, afin de remettre le système à l'heure:

mm= mois
dd= jour
HH= heure
MM= minutes
yy= année facultative

Le calendrier fourni par cal concerne un mois ou une année donnés:

mm= mois facultatif
yyyy= année facultative

Sans option, cal fournit le calendrier du mois courant de l'année actuelle.

Remarque: Les commandes UNIX disposent en général de dizaines d'options que nous ne saurions lister ici. Consulter le manuel pour plus de détails.

3) Le système de fichiers.

a) Architecture:

Le système de fichiers d'UNIX est hiérarchisé selon une arborescence de répertoires.
Le répertoire racine (root) / est le répertoire de plus haut niveau; il contient tous les autres.

Les répertoires et sous répertoires s'imbriquent sur plusieurs niveaux:

On nomme un répertoire à partir de la racine en séparant les intermédiaires par des /.

Exemples:
/
/tmp
/usr/bin
/home/maths/gilles
...

Chaque répertoire contient une multitude de fichiers qui sont propriété du système
ou des utilisateurs.

Remarque: Un système UNIX comporte des centaines ou des milliers de répertoires
et des milliers voire des dizaines de milliers de fichiers !!!

Un nom de fichier ou de répertoire peut avoir en général jusqu'à 255 caractères,
minuscules, majuscules, chiffres et de nombreux caractères spéciaux.

On accède à un fichier depuis la racine en ajoutant son nom au chemin d'accès
au répertoire où il se trouve:

Exemples:
/usr/bin/prog
/home/maths/gilles/algorithme2.c

Il existe différents types de fichiers sous UNIX:

- Les fichiers réguliers (Sources C, exécutables, fichiers de traitement de texte ...)
- Les drivers de périphériques (Fichiers situés dans /dev qui identifient un terminal ou un périphérique et permettent de communiquer avec lui)

- Les tubes fifo (Fichiers spéciaux de type first-in first-out)
- Les liens (Alias sur des fichiers - Voir plus loin)
- Les répertoires (Ils sont assimilés à des fichiers particuliers)

b) droits d'accès:

Chaque fichier du système (régulier ou non) a un propriétaire unique qui peut être le système (sys), l'administrateur (root), ou un utilisateur donné.
Chaque fichier dispose de droits d'accès pour

- le propriétaire,
- un groupe avec lequel le propriétaire peut accepter de partager son fichier,
- les autres.

Pour chacune de ces catégories, les droits d'accès sont au nombre de trois, soit neuf droits d'accès en tout:

r droit d'accès en lecture (read)
w droit d'accès en écriture (write)
x droit d'accès en exécution ou exploration

La commande ls liste les fichiers d'un répertoire:

Syntaxe: ls
ls *repertoire*
ls -l -a *repertoire*

ls tout court donne la liste des fichiers.
L'option -a entraîne l'affichage des fichiers habituellement cachés, dont le nom commence par un point.
L'option -l donne un affichage très détaillé comportant pour chaque fichier:

- le type de fichier
- les droits d'accès
- le nombre de liens
- le propriétaire
- le groupe d'appartenance
- la taille en octets
- la date de dernière modification
- le nom du fichier

Exemple:

```
$ ls ↵
essai.c
essai
projet_1995
$ ls -1↵
-rwxr-xr--, 1   gilles,   maths,  15204  Sep 15  10:05  essai
-rw-r--r--, 1   gilles,   maths,   9732  Sep 15  10:02  essai.c
drwxr-xr-x, 1   gilles,   maths,    512  Jul  05  15:32  projet_1995
```

Analysons les exemples issus de ls -l

- Le premier caractère indique le type de fichier selon la nomenclature suivante:
  - `-` pour un fichier régulier
  - `d` pour un répertoire
  - `b ou c` pour un driver de périphérique (bloc ou caractère)
  - `p` pour un tube fifo (pipe)
  - `l` pour un lien (link)

- Les neuf lettres suivantes sont les droits d'accès rwx rwx rwx mentionnés plus haut; les trois premiers sont les droits du propriétaire, puis les droits du groupe d'appartenance, et enfin les droits accordés aux autres utilisateurs.
  Un droit inhibé est remplacé par un -

  Ainsi rw- r-- r-- pour le fichier régulier essai.c signifie droit en lecture pour tous, mais en écriture —modification—pour le propriétaire seulement.

  Pour le répertoire projet_1995 les droits rwx r-x r-x signifient droit en lecture pour tous, mais droit en écriture—création ou suppression de fichiers— pour le propriétaire seulement et droit en exploration pour tous.

- Vient ensuite le nombre de liens associés au fichier: voir commande ln.

- Le propriétaire du fichier, ici gilles.

- Le groupe d'appartenance du fichier, auquel s'appliquent les droits correspondants.

- La taille du fichier en octets: 15204 pour essai.

- La date de création ou de dernière modification.

- Et enfin seulement le nom du fichier concerné !

La commande ls supporte les jokers * et ? qui permettent de n'obtenir que la liste des fichiers correspondant à un certain format:

*	remplace tout groupe de caractères
?	remplace tout caractère

Exemples:		
	ls -l *.c	donne la liste des fichiers terminés par .c
	ls -l essai*	donne la liste des fichiers commençant par essai...
	ls projet_????	donne la liste de tous les fichiers commençant par projet_... suivi de quatre caractères.

c) Les répertoires:

Un utilisateur se trouve à tout instant dans un répertoire donné de l'arborescence, appelé répertoire courant.
Lors de la connexion le répertoire courant est le répertoire de travail Home de l'utilisateur, qui lui a été attribué par l'administrateur; il s'agit en général du répertoire /usr/nom_groupe/login_name ou encore /home/nom_groupe/login_name.

Exemples:
/usr/maths/gilles
/home/maths/gilles

Le répertoire courant peut être obtenu avec pwd (print working directory) et on en change avec chdir ou cd (change directory).

Syntaxe:
pwd
chdir *repertoire*
cd *repertoire*

La commande pwd n'a pas de paramètre; elle affiche le nom complet du répertoire courant.
Par contre chdir ou cd reçoit le nom du répertoire dans lequel on veut passer; ce répertoire peut être absolu (depuis la racine /) ou relatif (par rapport au répertoire actuel)
s'il s'agit par exemple de descendre dans un répertoire en dessous ou de remonter au répertoire immédiatement au dessus.

Exemples:
chdir /usr/bin
cd /etc
chdir projet_1995
cd ..

Le raccourci .. désigne toujours le répertoire immédiatement au dessus.

Si on était dans /home/maths/gilles on passerait alors dans /home/maths.

On peut créer ou supprimer un répertoire—à condition que les droits d'accès le permettent—avec mkdir (make directory) ou rmdir (remove directory):

Syntaxe: mkdir *repertoire*
rmdir *repertoire*

Remarque: On ne peut supprimer qu'un répertoire vide de tout fichier.

d) Les fichiers réguliers:

Des opérations élémentaires sur les fichiers réguliers sont naturellement la copie, le déplacement, la suppression.

Ces opérations sont du ressort des commandes cp (copy), mv (move), rm (remove):

Syntaxe: cp *rep_srce/fichier_srce rep_dest/fichier_dest*
mv *rep_srce/fichier_srce rep_dest/fichier_dest*
rm *repertoire/fichier*

Si *rep_srce* ou *rep_dest* sont absents l'opération a lieu dans le répertoire courant.
Ces fonctions supportent les jokers * et ? dans les noms de fichier.

💣 Attention: Un fichier supprimé par rm est en général irrécupérable, même pour l'administrateur du système.
La commande rm * est de ce fait extrêmement dangereuse !
Elle demande parfois confirmation de la destruction de tous les fichiers du répertoire courant, mais ce n'est pas systématique.

e) Les liens:

On vient de voir que cp copie un fichier d'un répertoire dans un autre répertoire et éventuellement sous un autre nom, mais cela n'est pas toujours l'idéal:
supposons que l'on doive partager un fichier avec un collègue, chacun devant pouvoir le consulter ou le mettre à jour;
chacun peut bien sûr travailler sur sa propre copie du fichier mais alors les mises à jour faîtes par l'un ne seront pas visibles par l'autre !
Ils devraient donc travailler sur un seul et même fichier, mais dans ce cas l'un des deux au moins devra faire des acrobaties pour accéder audit fichier dans un répertoire lointain...

Il existe une solution à ce dilemme: la notion de lien.
Un lien est une entrée dans un répertoire qui fait référence à un fichier ou un répertoire situé en réalité ailleurs.
Un lien est crée par la commande ln (link)

Syntaxe: ln -s *filename linkname*
ln -s *dirname linkname*

Les opérations effectuées sur le lien *linkname* seront en fait réalisées sur l'original *filename* ou *dirname.*

Exemple: Dans notre problème gilles et dupont veulent partager le fichier data.bzh qui est actuellement dans /home/info/dupont.
Il suffit à gilles de créer le lien suivant, et à dupont de donner des droits d'accès suffisants à data.bzh:

ln -s /home/info/dupont/data.bzh data.bzh

Un lien sur un répertoire constitue par ailleurs un raccourci vers ce répertoire:

Exemple: Nous sommes dans /home/maths/gilles et nous créons le lien:

ln -s /usr/bin ubin

Alors le fait de passer dans ubin avec cd ubin nous fait passer en réalité dans /usr/bin comme on peut le vérifier avec pwd !

Lors de la création d'un lien, le nombre de liens du fichier original augmente d'une unité.
Pour supprimer un fichier il faut veiller à supprimer tous les liens dessus.

Chaque entrée .. du système de fichier est en fait un lien sur le répertoire au dessus.

f) Modifier les droits d'accès, le propriétaire, le groupe:

Seul le propriétaire d'un fichier—ou l'administrateur—peut en changer les droits d'accès à l'aide de chmod.

Syntaxe: chmod ??? *filename*
chmod u,g,o,a +-= r,w,x *filename*

chmod supporte les jokers * et ? dans les noms de fichiers.
Il y a en fait deux façons de procéder:

- **Le codage octal** (en base 8)

Les droits rwx rwx rwx sont codés en binaire par 1 pour un droit activé
et 0 pour un droit inhibé: ainsi rwx r-x --x donne le codage 111 101 001.
Ce mot binaire est ensuite codé en octal, chaque triplet représentant un chiffre
de 0 à 7; ici on obtient 751.

La commande chmod 751 *filename* octroie donc à *filename* les droits rwx r-x --x.

- **le codage littéral**

Cette fois on raisonne sur les droits du propriétaire (u), du groupe d'appartenance (g), des autres (o) ou de tous à la fois (a).
On ajoute (+), on enlève (-) ou on affecte (=) les droits rwx indiqués.

Exemples:	chmod o-wx toto	enlève à toto les droits wx pour les autres
	chmod a+x toto	ajoute pour tous le droit x
	chmod g=rx toto	spécifie les droits rx pour le groupe
	chmod u=rwx,g=rx,o=x toto	équivaut à chmod 751 toto

Chacune des deux approches présente des avantages selon ce que l'on veut obtenir.

Un fichier peut changer de groupe ou de propriétaire avec chgrp et chown.

Syntaxe: chgrp *group_name filename*
chown *user_name filename*

Ces deux commandes supportent les jokers * et ? dans les noms de fichiers.

Attention: Donner c'est donner !
Un fichier donné à un autre utilisateur ne peut plus être repris ! Seul le nouveau propriétaire ou l'administrateur peut vous le rendre.

g) La commande umask:

Les droits d'accès à un fichier nouvellement crée (par un éditeur ou autre) sont soumis à quelques limitations;
la commande umask détermine ces limitations pour chaque utilisateur:

Syntaxe: umask
umask ???

umask tout court affiche le masque de création de fichier actuel sont forme octale comme pour chmod.
Suivi de trois chiffres en octal, umask définit le nouveau masque, en indiquant les droits qui doivent être inhibés lors de la création d'un fichier: droits= droits & (~umask)

Exemple: umask 023

Les fichiers crées par l'utilisateur auront au plus les droits rwx r-x r--

h) Recherche de fichiers:

Dans les milliers de fichiers et répertoires du système UNIX il est indispensable de disposer d'un utilitaire de recherche de fichiers: find.

Syntaxe: find *origine* -name *filename* -type *lettre* -print

find recherche les fichiers de nom *filename* récursivement dans tous les sous répertoires à partir du répertoire *origine*.

L'option -print force l'affichage des fichiers trouvés.
L'option -type permet, suivie d'une *lettre* f, d, c, b, p de ne rechercher que les fichiers réguliers, répertoires, drivers caractères ou bloc, pipes.

find supporte les jokers * et ? à condition de les utiliser entre guillemets, ainsi que de nombreuses autres options qu'il serait trop long de lister ici !

Exemples: find / -name gilles -type d -print
find /home/maths/gilles -name "*.c" -print

Signalons une autre commande utile, grep, qui est capable de rechercher dans un fichier texte toutes les occurrences d'une expression donnée:

Syntaxe: grep -n '*expression*' *filename*

Le nom de fichier *filename* peut comporter des jokers * et ?
l'*expression* entre guillemets peut prendre des formes sophistiquées;
consulter le manuel à ce sujet.

L'option -n force la numérotation des lignes trouvées, afin de pouvoir les retrouver plus facilement ensuite dans le fichier.

Exemples: grep -n '#define' *.c
grep -n 'alpha' tempo.txt

i) Les principaux répertoires du système:

Tous les systèmes UNIX comportent des répertoires incontournables, dont le contenu est caractéristique:

/	la racine, réservée au kernel du système.
/bin	le répertoire des exécutables système.
/usr ou /home	le répertoire principal des utilisateurs.
/usr/bin	le répertoire des utilitaires

/dev	le répertoire des drivers de périphériques.
/etc	le répertoire des fichiers du système.
/tmp	le répertoire des fichiers temporaires.

4) Le shell et l'environnement:

Le shell est l'environnement de travail de chaque utilisateur. Il intervient dès la connexion et se manifeste par un prompt '$' ou '*' ou encore '%' en l'attente d'une commande.

Le shell permet diverses choses:

- Le dialogue entre l'utilisateur et le système d'exploitation, en lançant les commandes demandées.
- Le lancement de programmes exécutables, d'applications, en avant plan ou en tâche de fond.
- L'exécution de scripts shell permettant d'automatiser certaines tâches.
- La gestion des variables d'environnement.

Il existe désormais plusieurs shells, le shell de Bourne (sh), le C-shell (csh), le shell de Korn (ksh), et le shell restreint (rsh) notamment.
Le shell attribué à chaque utilisateur dépend de l'administrateur du système.

Les différents shells apportent les mêmes fonctionnalités—sauf le shell restreint qui est bridé pour des raisons de sécurité—mais diffèrent par leur langage de programmation
des scripts, leurs fichiers d'initialisation et leurs variables d'environnement.
Nous discuterons des spécificités au fur et à mesure.

a) Les commandes de base:

De nombreuses commandes ont déjà été vues; ajoutons y la commande clear qui efface l'écran, cat qui permet de visualiser un fichier texte, stty qui détermine les caractéristiques du terminal et man qui appelle le manuel d'aide intégré.

Syntaxe: clear
cat *filename*
stty -a

man -a *item*

La commande clear efface simplement l'écran.
L'utilitaire cat affiche le contenu du fichier texte *filename*, mais un problème se pose lorsque le fichier est trop grand car alors les pages défilent trop vite sans qu'on ait le temps de les étudier.
Une parade consiste à utiliser le filtre d'affichage <u>more</u> qui fractionne l'affichage en pages séparées par une pause:

<u>Exemple:</u> cat toto.c ¦ more

Un filtre est introduit à la fin d'une commande par le caractère ¦ et modifie la sortie de cette commande à l'écran.
Ici le filtre more assure un affichage paginé où l'utilisateur appuie sur une touche pour passer à la page suivante.

Signalons également le filtre <u>sort</u> qui permet de trier par ordre alphabétique les résultats d'une commande.

Par ailleurs stty -a dresse la liste des paramètres de fonctionnement du terminal.
On y trouve les paramètres et la vitesse de communication lorsqu'il s'agit d'un terminal série et —entre autres— la définition de plusieurs fonctions du terminal:

Action	**Usage**	**Touches associées**
intr	Break: interrompt une commande	Break ^C
erase	Backspace: efface un caractère erroné par retour arrière	^H ^ ?
kill	efface toute la commande en cours d'édition	^U
eof	demande de déconnexion	^D
susp	suspension d'une tâche qui est mise en attente	^Z

Les touches associées varient d'un système à l'autre et ne correspondent pas forcément à celles indiquées ici.
Le rôle de la commande stty -a est justement d'indiquer les combinaisons en vigueur.
stty permet également de modifier les dispositions en vigueur:

Exemple: stty erase ^H

Permet de redéfinir la combinaison associée à l'action erase.

La plus importante est assurément intr (^C) qui permet d'interrompre la commande en cours,
lorsque celle-ci dure trop longtemps ou a induit un blocage.

Exemple: Une recherche par find telle que find / -name *.c -print
peut très bien continuer sa recherche alors que le fichier voulu a déjà été localisé.
On utilise alors intr (^C) pour interrompre la suite de la recherche.

L'utilitaire man est d'un grand secours devant la multitude de commandes et d'options que propose UNIX. man accède à une base de données contenant la description détaillée de chaque commande ou fonction du langage C, telle qu'implémentée sur le système en question:

Exemple: man -a find
man -a printf

Le premier appel expose le rôle de find et décrit les multiples options disponibles.
Le second donne la syntaxe de la fonction printf( ) du langage C et la façon de l'employer.

L'option -a est facultative mais conseillée dans la mesure où le manuel est scindé en plusieurs sections; l'option -a force alors la recherche dans toutes lesdites sections.

Pour avoir la liste des sections faire tout simplement man man !

b) L'environnement:

Le shell mémorise le répertoire courant mais aussi des variables d'environnement utiles, voire indispensables; ces variables sont en Majuscules pour les Bourne shell et Korn shell et en minuscules pour le C-shell

La liste des variables d'environnement est fournie par set:

Syntaxe: set
set *var=chaine*
VAR=chaine

Sans paramètre le résultat obtenu est une liste de variables contenant chacune

une chaîne de caractères.

Exemple: Bourne shell ou Korn shell

```
$ set ↵
PATH=/bin:/usr/bin:.
HOME=/home/maths/gilles
TERM=vt220
...
$
```

C-shell

```
% set ↵
path=/bin:/usr/bin:.
home=/home/maths/gilles
term=vt220
...
%
```

Dans la longue liste de variables définies, les plus importantes ont été retenues:

PATH ou path   Contient la liste des répertoires dans lesquels les commandes, utilitaires, exécutables sont recherchés.
En effet un exécutable du répertoire courant ne sera pas trouvé si le répertoire courant . ne figure pas dans le Path.

HOME ou home   Répertoire Home de l'utilisateur.
On peut y faire appel comme à toute variable d'environnement en la faisant précéder d'un $

Exemple: cd $HOME
cd $home
Opère un chdir vers /home/maths/gilles

TERM ou term: Précise le type de terminal utilisé.
Voir à ce sujet le chapitre consacré aux terminaux.
S'il y a une chose que le système ne peut pas deviner, c'est bien le modèle de terminal qui est au bout du fil !
Ici on déclare qu'il s'agit d'un DEC vt220 ou compatible.

On peut afficher une variable d'environnement avec echo:

<u>Syntaxe:</u> echo 'message'

<u>Exemple:</u> echo 'Mon terminal est un $TERM'
echo 'Mon terminal est un $term'

Affichera le message Mon terminal est un vt220, après substitution de la variable par son contenu.

La commande set permet aussi, sous la seconde forme, de modifier une variable d'environnement C-shell; avec les autres shells on déclare ou définit directement les variables:

<u>Exemple:</u> set term=vt100 (C-shell)
set path=$path:/usr/local/bin
TERM=vt100 (Bourne ou Korn shell)
PATH=$PATH:/usr/local/bin

Cela modifie la définition du terminal, qui est désormais supposé être un DEC vt100. La seconde commande ajoute un répertoire au chemin d'accès Path.

Les autres variables d'environnement sont des indications destinées à certaines applications (mail, vi) ou à des programmes spécifiques...

Les variables d'environnement sont définies dans le shell actif; pour qu'une telle variable soit transmise à un sous shell ou à un programme appelé par le shell il est nécessaire d'avoir recours à <u>export</u> (sh ou ksh) ou <u>setenv</u> (csh):

<u>Syntaxe:</u> export *VAR*
setenv *var*

La commande export *VAR* suit en général une commande *VAR=chaine* et assure l'exportation
de la variable ainsi définie; coté C-shell, setenv *var* remplace complètement set *var* en assurant la définition et l'exportation de la variable.

<u>Exemples:</u> HELP=toto.help (Bourne ou Korn shell)
export HELP

setenv help=toto.help (C-shell)

<u>c) Les scripts de configuration:</u>

Chaque utilisateur trouvera dans son répertoire Home un fichier .profile (sh ou ksh)

ou deux fichiers nommés .login et .cshrc (csh).

Ces fichiers de configuration sont exécutés par le shell lors de la connexion;
ils contiennent des commandes et la définition de diverses variables d'environnement.

L'utilisateur peut éditer et modifier ces fichiers de configuration.
Pour en comprendre les subtilités se reporter au chapitre consacré à la programmation des scripts shell.

d) La redirection des entrées/sorties

Par défaut le shell et les commandes qu'il lance prennent leurs données (entrée) au clavier et envoient leurs résultats (sortie) à l'écran, via le driver de périphérique associé au terminal /dev/tty...

En fait, sous UNIX, un périphérique est vu comme un fichier spécial, et ce qui peut être envoyé vers un périphérique peut aussi être envoyé dans un fichier et vice versa.

On peut rediriger la sortie standard d'une commande avec >
Pour rediriger la sortie erreur standard—qui reçoit les messages d'erreur d'une application, envoyés également par défaut à l'écran—on utilisera 2>

Syntaxe: commande > *filename*
commande 2> *filename*

Exemple: ls -l > listing

Cette commande redirige la sortie standard de ls vers le fichier listing, qui sera crée pour la circonstance et qui contiendra l'équivalent de ce que ls -l aurait affiché à l'écran.
On le vérifiera avec cat listing.

Remarque: /dev/null est un périphérique spécial qui joue le rôle de gouffre sans fond dont rien ne ressort. Tout ce qui y est envoyé est ignoré.

L'entrée standard, la sortie standard et la sortie erreur standard sont accessibles par les numéros (handles) 0, 1 et 2, ce qui explique la commande suivante.

Exemple: commande 2> /dev/null

Tous les messages d’erreur de la commande sont redirigés dans la poubelle, évitant ainsi des affichage incongrus (parfois utile dans des scripts).

On peut aussi rediriger la sortie d'une commande en l'ajoutant à un fichier existant avec >>

<u>Syntaxe:</u> commande >> *filename*

<u>Exemple:</u> stty -a >> listing

Ajoute au fichier listing le résultat de la commande stty -a

Enfin, si on souhaite conserver l'affichage tout en dupliquant la sortie on utilisera le filtre <u>tee</u>:

<u>Syntaxe:</u> commande ¦ tee *filename*

Le résultat de la commande est envoyé à l'écran et dans le fichier *filename*

Signalons que < permet de rediriger, ce qui est moins fréquent, l'entrée standard d'une commande depuis un fichier ou un périphérique.

<u>Syntaxe:</u> commande < *filename*
commande < */dev/terminal*

5) L'éditeur vi:

UNIX ne serait plus UNIX sans l'éditeur de texte **vi** (visual).
Certes assez puissant, cet éditeur, qui a dû être en son temps révolutionnaire,
souffre aujourd'hui d'une ergonomie préhistorique ! Il est néanmoins incontournable
car c'est le seul éditeur que l'on soit certain de trouver sur tout système UNIX.
On trouvera souvent aussi l'éditeur **emacs** du GNU, mais en mode texte il est
tout aussi préhistorique que vi faute de menus déroulants.

Syntaxe: vi
vi *filename*

L'éditeur vi a deux modes de fonctionnement, entre lesquels on ne cesse de commuter:

- le mode commande
- le mode édition

Au lancement, vi est en mode commande; une commande est pour lui une lettre
ou une séquence de lettres suivie éventuellement d'options...

On revient en mode commande en pressant la touche ESC. Un beep sonore
signifie que l'on était déjà en mode commande. (C'est très utile car souvent on ne sait
même plus dans quel mode on est !)

On passe en mode édition avec l'une des commandes d'édition ci-après;
dans ce mode seulement on peut saisir le texte voulu ou modifier le texte existant,
au niveau du curseur d'affichage.

Certaines commandes particulières sont introduites par : ou par /
Dans ce cas le curseur d'affichage passe sur la dernière ligne de l'écran, appelée ligne
de commande, : ou / s'y affiche en attendant le reste de la commande terminée par ↵

Commandes de déplacement:

En général les flèches ← ↑ ↓ → et éventuellement PgUp et PgDn permettent
de déplacer le curseur d'affichage.
Une tentative de déplacement hors limites du texte ramène vi en mode commande.

Commandes d'édition:

Toute saisie ou modification de texte commence par l'une des commandes suivantes:

(ESC) i	insert	Insertion de texte à la position du curseur
(ESC) a	append	Insertion de texte après le curseur ou en fin de ligne

(ESC) o	open	Ouvre une nouvelle ligne sous le curseur et place le curseur au début de celle-ci.
(ESC) x	delete	Efface le caractère sous le curseur.
(ESC) dd	delline	Efface toute la ligne sous le curseur.
(ESC) yy	copy	Copie la ligne courante dans le tampon d'édition.
(ESC) p	paste	Copie la ligne présente dans le tampon d'édition sous la ligne courante.

Remarque: Il existe bien d'autres commandes permettant notamment de manipuler des blocs de texte, mais on entre là dans l'univers effroyable des commandes sophistiquées de vi.

Commandes de recherche:

^G	Affiche le numéro de la ligne courante sur la ligne de commande.
(ESC) / *chaine*	Recherche d'une chaîne en avant dans le texte.
(ESC) ? *chaine*	Recherche d'une chaîne en arrière dans le texte.
(ESC) n	Poursuivre la recherche dans le même sens.
(ESC) N	Poursuivre la recherche en sens inverse.

Commandes de sauvegarde et de sortie:

(ESC) :w *filename* ↵	Ecriture du texte dans le fichier *filename*. Si *filename* est omis le fichier est sauvegardé sous son nom actuel.
(ESC) :x *filename* ↵	Idem mais en quittant vi après la sauvegarde.
(ESC) :q ↵	Quitter vi.
(ESC) : q! ↵	Quitter vi et forcer l'abandon des modifications.
(ESC) :α,βw *filename* ↵	Sauver les lignes α à β dans le fichier *filename*.
(ESC) :r *filename* ↵	Insérer le contenu du fichier *filename* à partir de la ligne courante.

Bonne chance dans l'utilisation de vi ! Il vous faudra un peu d'entraînement je crois !

💣 Attention: vi fonctionne en mode plein écran; afin qu'il fonctionne correctement il est indispensable que la variable TERM ou term soit correctement renseignée
et indique le terminal ou l'émulation effectivement utilisé.

6) Le gestionnaire d'impression lp:

a) Requêtes d'impression:

Comme dans tout système multitâche et multiutilisateur, nul ne peut disposer à son gré des imprimantes du système. Les impressions passent donc par un gestionnaire d'impression—spooler—nommé lp.
Il reçoit les requêtes d'impression des utilisateurs, les mémorise dans une file d'attente et les répercute sur les imprimantes disponibles dès que possible.

Pour tout problème lors de l'impression consulter l'administrateur du système.

Syntaxe: lp *filename*
lp -d*printer filename*
lp -*nb filename*

lp *filename* envoie le fichier *filename* au gestionnaire pour impression.
L'option -d*printer* permet de choisir explicitement l'imprimante destination, pour des raisons de qualité d'impression ou de proximité par exemple.
Pour obtenir la liste des imprimantes disponibles, voir plus loin.

L'option -*nb* où *nb* est un nombre permet d'obtenir plusieurs copies du fichier.

lp répond à une requête en affichant le numéro d'ordre attribué à cette requête, ou un message d'erreur en cas d'impossibilité d'accéder à la requête.

Exemple:

```
$ lp -d epsonlq source.c ↵
request id is epsonlq-5 ( 1 file )
$
```

b) Suivi des requêtes:

Une impression peut être sérieusement différée dans le temps si de nombreux utilisateurs veulent imprimer au même moment.

Pour savoir où en est une requête on peut utiliser lpstat.

Syntaxe: lpstat -v
lpstat -u *user_name*

lpstat -t

Avec l'option -v lpstat affiche la liste des imprimantes actives du système.
Avec l'option -u *user_name* un utilisateur peut avoir un aperçu de l'état de ses propres requêtes d'impression; une requête qui n'apparaît plus dans la liste a été traitée et il ne reste plus qu'à récupérer les documents sur l'imprimante.
Enfin l'option -t dresse un état complet du système d'impression.

On peut stopper une requête d'impression avant qu'elle n'ait été traitée avec <u>cancel</u>:

<u>syntaxe:</u> cancel *request_id*

On ne passe pas à cancel un nom de fichier car un tel renseignement serait insuffisant; on doit lui communiquer le numéro de requête tel que formulé par lp.
On peut le récupérer si nécessaire avec lpstat.

<u>Exemple:</u>

```
$ lpstat -u gilles ↵
epsonlq-5       gilles    3251    Sep 15 11:13   on epsonlq
$ cancel epsonlq-5 ↵
request epsonlq-5 cancelled
$
```

7) Gestion des processus:

a) Généalogie:

Un **processus** est un programme en cours d'exécution sur le système.
UNIX étant multitâche et multiutilisateur, plusieurs processus s'exécutent au même moment, appartenant à différents utilisateurs.
En fait un seul processus—le processus élu—est en cours d'exécution à un instant donné, mais un mécanisme complexe et très rapide de commutation des tâches crée l'illusion auprès des utilisateurs.

Au démarrage du système il y a un processus, init, qui assure le lancement des processus vitaux pour le système, selon la configuration définie par l'administrateur:

- vérification de l'intégrité du système
- démarrage des gestionnaires de réseau
- démarrage du gestionnaire d'impression
- démarrage du gestionnaire de mail
- démarrage des mécanismes de login sur les terminaux
- démarrage d'X-Window si nécessaire
- ...

Un processus peut déclencher l'exécution d'autres processus et ainsi de suite...
Il y a donc à chaque instant une arborescence des processus actifs, dont la racine init est l'ancêtre commun à tous.

Exemple:

Le shell lui-même, en fin de liste, passe son temps à examiner les commandes qui lui sont soumises et à exécuter les processus correspondants.

Chaque processus du système est identifié de façon unique par un numéro nommé pid (process identifier); il connaît aussi le pid de son père (celui qui l'a engendré) nommé ppid (parent pid).

Pour obtenir des informations sur les processus en cours on fait appel à ps:

Syntaxe:

ps
ps -f
ps -u *user_name*
ps -t */dev/terminal*
ps -eaf

Tout court, ps donne la liste des processus de l'utilisateur courant.
L'option -f y ajoute une multitude de détails supplémentaires.

L'option -u *user_name* dresse la liste des processus d'un utilisateur donné.
Avec l'option -t */dev/terminal* on obtient la liste des processus rattachés à un terminal de contrôle donné, c'est à dire lancés depuis ce terminal.

Enfin l'option -eaf dresse la liste exhaustive de tous les processus du système.

Exemples:

```
$ ps ↵
PID    TTY    TIME  COMMAND
392     03    0:01  ksh
464     03    0:00  ps
$ ps -f ↵
UID     PID   PPID  C    STIME      TTY   TIME  COMMAND
gilles  392   1     1    09:32:07    03   0:01  -ksh
gilles  475   392   9    09:47:01    03   0:00  ps -f
$
```

Commentaires:

Dans la première commande, ps, on voit que le shell ksh a un pid 392, qu'il s'exécute depuis le terminal tty03, qu'il a consommé un temps système de 1/100 de seconde.

Dans la seconde commande, ps -f, on a plus de détails:
ksh est précédé d'un tiret - qui signifie qu'il s'agit du shell de connexion.
Il est le fils du processus de pid 1 (ppid) c'est à dire init, il a démarré à 09h32mn07s avec un niveau de priorité C=1.

Dans cette seconde commande, ps n'a pas le même pid que dans la première car il s'agit de deux commandes différentes !

Les pid sont attribués par le kernel du système par ordre de numéros croissants:

les pids les plus petits correspondent aux processus les plus anciens.
Le pid d'un processus terminé n'est pas réutilisé.

b) Environnement et héritage d'un processus:

Chaque processus en cours d'exécution dispose d'un certain nombre d'informations:

- le nom du propriétaire de ce processus et son groupe d'appartenance
- un environnement
- une liste de paramètres transmis par le père
- un terminal de contrôle
- un répertoire de travail
- une table des descripteurs de fichiers ouverts
- un masque de création de fichiers
- une priorité d'exécution

Le processus hérite une bonne partie de ces informations de son père, qui souvent est le shell ayant exécuté la commande.

L'environnement est constitué par la liste des variables d'environnement telles que PATH, TERM, HOME ou path, term, home; en plus de ces variables standard le père peut transmettre à son fils des variables d'environnement supplémentaires.

Les paramètres sont ceux transmis en général sur la ligne de commande.

Le terminal de contrôle est le terminal d'où a été lancé le processus.

Chaque processus a un répertoire de travail—répertoire courant—qu'il hérite de son père et qu'il peut ensuite changer à sa guise.

La table des descripteurs de fichiers ouverts gère les fichiers manipulés par le processus;

il y a toujours au moins trois fichiers dans cette table, qui font référence à l'entrée standard, la sortie standard et la sortie erreur standard.
Un fils hérite en fait de tous les descripteurs sur les fichiers ouverts par son père; toutes les manipulations que fera l'un des deux processus sur un de ces fichiers sera visible de l'autre processus.
Par contre un fichier nouvellement crée par le fils n'est pas automatiquement accessible au père.

Le masque de création de fichiers—umask—sert à déterminer les droits d'accès à tout fichier nouvellement crée; on peut modifier ce masque.

Remarque: Nous verrons au niveau programmation comment agir sur toutes ces propriétés d'un processus.

c) Processus en tâche de fond:

Jusqu'à présent nous avons lancé des commandes en **avant-plan**, c'est à dire que le shell s'effaçait temporairement, laissait la commande s'exécuter et reprenait la main une fois cette commande terminée.

Une autre possibilité consiste à exécuter un processus en **tâche de fond** :
dans ce cas le processus est lancé et le shell reprend la main aussitôt.
Bien sûr cela ne peut convenir qu'à un processus non interactif, car si le processus prétend saisir ou afficher des données, il va y avoir conflit avec le shell pour l'accès au clavier et à l'écran !
C'est un bon moyen d'exécuter des programmes purement calculatoires, éventuellement assez longs, tout en continuant à faire autre chose !

Syntaxe: *Commande* &

Un & final provoque l'exécution de la *commande* en tâche de fond.
Le shell affiche alors le pid de la tâche de fond ainsi crée, puis il reprend la main en attente d'une autre commande avec le prompt habituel.

On peut prendre des nouvelles d'une tâche de fond avec ps:
tant que la tâche apparaît dans la liste c'est qu'elle n'est pas terminée.

La commande wait permet de suspendre le shell jusqu'à ce que certaines tâches de fond soient terminées:

Syntaxe: wait
wait *pid*

wait sans paramètre suspend le shell jusqu'à la fin de toutes les tâches de fond.
Avec un *pid* particulier, le shell est suspendu jusqu'à la fin du processus indiqué.

Si on veut abandonner une tâche de fond devenue inutile, il est possible de tuer
le processus en question par la commande kill.

Syntaxe: kill -15 *pid*
kill -9 *pid*

On indique à kill le *pid* du processus visé et une option numérique indiquant quel
signal doit être envoyé audit processus:

SIGTERM=15 Signal de terminaison.
SIGKILL=9 Signal de terminaison inconditionnelle
plus puissant que le précédent.

Remarques: Un utilisateur ne peut tuer que des processus lui appartenant.
L'administrateur peut tuer tout processus.
Regardez ce qui ce passe lorsque vous tuez votre propre shell !
Une tâche de fond peut présenter des avantages, mais il faut savoir qu'à la déconnexion
toutes les tâches de fond seront tuées par le système...
Que faire si on veut lancer un programme le soir et récupérer les résultats le lendemain
sans passer la nuit sur son terminal ?

Il suffit d'utiliser nohup (no hangup) !

Syntaxe: nohup *commande*

Dans ce cas la *commande* continuera de s'exécuter après la déconnexion,
ignorant le signal SIGUP.
La sortie standard ne pouvant plus se faire à l'écran, celle-ci sera par défaut redirigée
vers le fichier nohup.out.
On utilise souvent nohup conjointement avec & pour réaliser une tâche de fond.

d) Suspension du processus en cours:

Lorsqu'une commande en avant plan laborieuse prend plus de temps que prévu
on peut:

- attendre patiemment
- la terminer brutalement avec le signal SIGINT (touche intr=^C)
- la suspendre temporairement avec le signal SIGTSTP (touche susp=^Z)

Consulter la commande stty pour les combinaisons associées.

Une fois le processus suspendu, le shell reprend la main en affichant le pid
de ce processus ou son numéro de job entre crochets [ ].

On pourra relancer ultérieurement ce processus avec le signal SIGCONT généré par le shell à la commande fg (foreground) ou bg (background):

Syntaxe: fg *pid* ou fg %*job*
bg *pid* ou bg %*job*

Remarque: Il semble que certains shell seulement proposent ces facilités permettant respectivement de relancer le processus en avant plan et en tâche de fond.
A défaut on pourra générer soi-même le signal SIGCONT avec la commande
kill -25 *pid*

e) Orphelins, démons et zombies:

Tout processus qui se termine renvoie par le biais du système un code de retour numérique à son père; le père peut en prendre connaissance dès que le fils est terminé.

Lorsque, pour une raison à préciser, le processus père est mort (terminé) avant son fils, le fils orphelin est adopté par le processus init qui devient son père adoptif.
C'est donc init qui recevra le code de retour.

Il arrive aussi que le père ne prenne pas la peine de consulter le code de retour de son fils; alors le processus fils est maintenu dans un état dit zombie , et il apparaît comme tel dans la liste des processus avec la mention defunct.

Signalons enfin que certains processus systèmes gérant le réseau, le mail ...
sont appelés démons (daemons); leur particularité est de n'être rattachés à aucun terminal de contrôle.

8) Programmation de scripts shell.

Les shells possèdent un langage de programmation permettant l'écriture de scripts afin d'automatiser certaines tâches...

Un tel script est un fichier texte composé de toutes sortes de commandes telles qu'on les utilise en mode interactif, liées par des instructions spécifiques de test, de saut ou de répétition.

Pour fonctionner, un script shell doit avoir le droit d'exécution x; il suffit alors de taper le nom *script_name* ↵ de ce script pour qu'il s'exécute.
En fait le shell principal génère un sous shell dédié à l'exécution du script, qui est alors interprété; cela permet aussi de tuer plus facilement la tâche ainsi générée en cas de besoin.

Hélas les langages de programmation des shells de Bourne et de Korn d'une part, et du C-shell d'autre part sont sensiblement différents; aussi traiterons nous la chose en deux parties.

Notons que l'on peut forcer l'exécution d'un script avec un shell particulier en lançant une commande *shell script_name*, où *shell* sera au choix sh, ksh ou csh.

a) programmation de scripts sous sh et ksh:

Remarquons que lors de la connexion les shells de Bourne et de Korn exécutent automatiquement les scripts d'initialisation suivants:

/etc/profile
.profile

Les commentaires

Les commentaires sont introduits dans un script shell par un #
Tout ce qui suit sur la ligne est ignoré par le shell lors de l'exécution.

Les variables

Un programme n'étant rien sans variables, le shell peut manipuler dans les scripts des variables numériques entières, des variables chaînes de caractères et des listes.
Une nouvelle variable est définie par:

variable=valeur	# variable numérique
variable=chaine	# variable chaîne
variable=chaine1 . . . chaine2	# variable liste

On accède par la suite au contenu d'une variable en faisant précéder son nom d'un $

comme pour les variables d'environnement, qui sont d'ailleurs accessibles
dans les scripts à condition d'avoir été exportées.
Une liste est une variable composée de plusieurs chaînes séparées par un délimiteur:
espace, tabulation, saut de ligne...

La saisie du contenu d'une variable s'opère à l'aide de read sous la forme:

read *variable*

L'affichage est du ressort de la commande echo sous la forme:

echo "texte $*variable*" ou echo texte $*variable*

Les caractères spéciaux suivants doivent être précédés d'un \ lorsque le texte
n'est pas entre guillemets: " ' ( ) \
ceci afin d'éviter leur interprétation par le shell qui leur attribue un sens particulier.

On mettra donc des guillemets de préférence dans les commandes echo !

Exemple de script sh ou ksh:

Saisir ce script sous vi, lui donner les droits d'exécution et le lancer.

```
# script essai pour sh et ksh

system=Sco_Unix            # sans espace derriere =

clear                      # efface l'ecran
echo "Quel est ton nom ?"
read name

date                       # affiche date et heure
echo "Bonjour $name"
echo "Bienvenue sur $system dans $HOME."
```

Un script peut définir ou supprimer des variables d'environnement (qui lui sont propres);
Il les crée comme d'habitude et les supprime à l'aide de unset:

VARIABLE_ENV=chaine
unset *VARIABLE_ENV*

<u>Remarque:</u> En fait les variables numériques sont stockées sous forme de chaînes de caractères par le shell; aussi le calcul avec ces variables requiert-il des techniques spéciales (voir plus loin).

La substitution de commandes

Un mécanisme très puissant permet de récupérer dans une variable le résultat qu'une commande afficherait à l'écran en temps normal.

Il suffit pour cela de placer la commande entre apostrophes inverses ` `:

chaine=`commande`
liste=`commande`

On obtient une chaîne ou une liste selon ce que retourne ladite *commande*.

<u>exemples</u>
workdir=`pwd`
# crée une chaîne contenant le répertoire courant
files=`ls`
# crée une liste des fichiers du répertoire courant, séparés par des espaces

Les listes ainsi crées seront utilisées dans des boucles for.

Les calculs numériques

Comme signalé plus haut, les calculs numériques sont compliqués par le fait que les variables sont mémorisées sous forme de chaînes.
L'utilitaire expr permet de s'en sortir, mais assez laborieusement:
expr affiche normalement à l'écran le résultat d'un opération sous la forme

expr *operande1 operateur operande2*

où l'opérateur peut être (entre apostrophes): '+' '-' '*' '/' '%'
et les opérandes sont des nombres ou des variables référencées par $*variable*.

<u>Exemples:</u>
expr 4 '+' 3 # affiche 7
expr $a '*' $b # affiche le produit de a par b

Afin de récupérer le résultat dans une variable, on doit faire une substitution de commande !

variable=`expr *operande1 operateur operande2*`

<u>Exemples:</u>

somme=`expr $a '+' $b`	# somme=a+b
carre=`expr $a '*' $a`	# carre=a*a
double='expr 2 '*' $a`	# double=2a

Les tests

On peut effectuer des tests sur les variables chaînes de caractères, les variables numériques, sur le code de retour des commandes et sur la nature des fichiers.
La syntaxe d'un test est au choix:

if [*test*]
then
    commandes
fi

if [*test*]
then
    commandes
else
    commandes
fi

💣 <u>Attention:</u> Les mots clés doivent être chacun sur une ligne.
Le *test* doit être séparé des crochets [et] par au moins un espace.

Un test sur le code de retour d'une commande prendra la forme suivante:

if [*commande*]

Il porte alors sur le code de retour—nul ou non nul—de ladite commande.

Un test numérique aura la forme suivante:

if [*operande1 operateur operande2*]

avec un opérateur parmi

- -eq # teste l'égalité d'*operande1* et *operande2*
- -ne # teste la non égalité d'*operande1* et *operande2*
- -lt # teste si *operande1* < *operande2*
- -gt # teste si *operande1* > *operande2*
- -le # teste si *operande1* ≤ *operande2*
- -ge # teste si *operande1* ≥ *operande2*

Exemple:

```
# script essai pour sh et ksh

echo "Entrer un nombre de 0 a 5:"
read chiffre

if [ $chiffre -gt 5 ]
then
	echo "Erreur"
else
	echo "Ok"
fi
```

Les tests sur chaînes de caractères auront deux formes possibles:

if [*testeur chaine*]
if [*chaine1 operateur chaine2*]

testeur étant au choix

-z	# teste si la chaîne est vide
-n	# teste si la chaîne est non vide

et *operateur* étant l'un des deux suivants

=	# teste l'égalité
!=	# teste la non égalité

Enfin un test sur la nature d'un fichier ressemblera à

if [*testeur filename*]

avec de nombreuses possibilités pour *testeur*

-f	# teste si le fichier est régulier
-d	# teste si le fichier est un répertoire
-c	# teste si le fichier est un driver caractère
-b	# teste si le fichier est un driver bloc
-p	# teste si le fichier est un tube
-r	# teste si le fichier est autorisé en lecture
-w	# teste si le fichier est autorisé en écriture
-x	# teste si le fichier est autorisé en exécution ou exploration
-s	# teste si le fichier est non vide

Les combinaisons de tests

Bien sûr on peut combiner plusieurs test avec les combinateurs -a (and) et -o (or)

sous la forme:

if [*test1 combinateur test2*]

Une négation s'introduit par !

if [! *test*]

On peut parenthèser les tests pour forcer la priorité, à condition de faire précéder chaque parenthèse d'un antislash: \(\)
pour éviter une interprétation abusive par le shell.

Remarque: Les tests entre crochets [ ] remplacent symboliquement un appel à l'utilitaire test qui renvoie un code nul ou non nul pour False et True.

On peut remplacer des tests multiples par un appel à case dont la syntaxe est:

case *variable* in
cas1) *commandes*;;
...
casN) *commandes*;;
esac

La *variable* est une chaîne de caractères. Chaque cas est terminé par deux points virgules. Les cas sont des chaînes auxquelles la *variable* peut s'identifier, comprenant éventuellement des jokers * et ?

On peut combiner deux cas en un seul avec ¦

Exemple:

```
# script essai pour sh et ksh

echo "Yes or No (Y-N) ?"
read reponse

case $reponse in
y|Y)    echo "Ok";;
n|N)    echo "Abandon"
        exit;;
esac
```

Les boucles

Il existe plusieurs types de boucles, while, until et for.

Les boucles while:

Leur syntaxe est while [*test*]
do
commandes
done

💣 Attention: Les mots clés doivent être chacun sur une ligne.

La boucle est répétée tant que le *test* est satisfait. Le *test* peut prendre toutes les formes évoquées plus haut.

Exemple:

```
# script essai pour sh ou ksh
echo "lister les carres jusqu'a combien ? "
read n
i=0

while [ $i -le $n ]
do
        carre=`expr $i '*' $i`    # calcul du carre
        echo $carre
        i=`expr $i '+' 1`         # incremente i
done
```

Les boucles until:
Leur syntaxe est until [*test*]
do
commandes
done

Le principe est comparable sauf que la boucle est répétée jusqu'à ce que le *test* soit enfin satisfait !

Les boucles for:
Leur syntaxe est ici for *variable* in *liste*
do
commandes
done

variable est une variable chaîne de caractères et *liste* une liste qui peut prendre plusieurs formes:

- liste explicite telle que
  liste= janvier février mars avril mai juin juillet août septembre

- liste implicite obtenue par substitution de commande telle que
  liste=`ls`

- liste de fichiers obtenue avec des jokers telle que
  liste=*.c

Exemple:

```
# script essai pour sh et ksh

rep=`pwd`
echo "repertoire: $rep"

for filename in `ls`
do
        if [ -f $filename ]
        then
                echo "$filename est un fichier "
        fi

        if [ -d $filename ]
        then
                echo "$filename est un repertoire"
        fi
done
```

break et continue:

Dans une boucle, les instructions break et continue ont pour effet respectif
de mettre fin à la boucle et de mettre fin à l'itération en cours.

La récupération des paramètres dans un script

Comme tout programme un script peut recevoir lors de son lancement des paramètres
sur la ligne de commande, séparés par des espaces.
Ces paramètres sont accessibles dans le script sous forme de variables prédéfinies
référencées alors par $0, $1, . . . , $9.

$0 est conventionnellement le nom du script, $1, . . . , $9 sont effectivement d'éventuels
paramètres transmis. $# est le nombre de ces paramètres sans compter $0.

Pour une manipulation globale on dispose encore des variables:
$* qui est la chaîne de tous les paramètres, utile pour retransmettre tous les paramètres
à une commande ou un autre script...
$@ qui est une liste de tous les paramètres utilisable en tant que liste.

💣 Attention: Un appel à set détruit paraît-il la liste des paramètres !

Exemples:

```
# liste des parametres recus
echo "script $0"
echo "liste des parametres:"

for p in $@
do
	echo $p
done
```

```
# script essai pour sh et ksh
# copie de sauvegarde des fichiers d'un repertoire en .bak
echo "liste des fichiers traites:"

if [ $# -eq 0 ]		# si aucun parametre transmis
then
	echo "Aucun fichier"
	echo "syntaxe: $0 repertoire"
else
	liste=`ls $1`
	for filename in $liste
	do
		echo "$1/$filename"
		cp $1/$filename $1/$filename.bak
	done
fi
```

b) Les spécificités du C-shell:

Le C-shell tient son nom de son langage de programmation des scripts qui se veut assez proche du langage C.

Un script pour C-shell commence impérativement par un # en début de la première ligne, donc par un commentaire.
A défaut le C-shell invoquera un sous shell sh et non csh pour exécuter le script.

Lors de la connexion le C-shell exécute lui-même les scripts d'initialisation suivants:

/etc/cshrc
.cshrc
.login

Lors de la déconnexion il exécute automatiquement et s'il existe le script

.logout

Les commentaires

Ils sont toujours introduits par un #

Les variables

Le C-shell distingue, lui, les variables numériques des variables chaînes de caractères.
On introduit une variable chaîne par set:

set *variable=chaine*

et une variable numérique par le caractère @:

@ *variable=valeur*

On doit mettre un espace entre @ et le nom de la variable.

La commande read ne permet plus la saisie des variables au clavier; elle est remplacée par la pseudo-variable $< qui représente une saisie au clavier:

set *variable*=$<
@ *variable*=$<

permettent respectivement de saisir une *variable* chaîne et une *variable* numérique.

Les calculs numériques

Les calculs numériques sont simplifiés et s'écrivent plus naturellement:

@ *variable=operande1 operateur operande2*

où les opérandes sont des nombres ou des variables numériques,
et l'opérateur est l'un de ceux-ci:

+ − * / % >> << & ¦ ^

On peut aussi abréger certaines opérations comme en langage C sous la forme:

@ *variable operateur=operande*

où *operateur=* est au choix += −= *= /= %=

Enfin on peut aussi utiliser ++ et −− sous la forme

@ *variable*++ ou @ *variable*−−

pour incrémenter ou décrémenter une *variable* numérique.

Les listes sont manipulées un peu différemment sous C-shell: on crée une liste par

set *liste*=(*liste de champs*)

On accède ensuite aux éléments de la *liste* à l'aide de leur indice, commençant à 1:

$*liste*[$*k*]   est le *k*-ième élément, où *k* est une variable numérique.
set *liste*[$*k*]=*chaine*   ou   @ *liste*[$*k*]=*valeur*
permettent de modifier un élément de la *liste*
$#*liste*   représente le nombre d'éléments de la *liste.*

La substitution de commandes

Voir plus haut; aucun changement à signaler.

Les tests

Le principe des tests est toujours le même, mais la syntaxe change:

```
if ( test ) commande    ou    if ( test ) then
                                     commandes
                              else
                                     commandes
                              endif
```

💣 Attention: Le then doit impérativement être sur la même ligne que le if.

Un test peut porter sur une chaîne de caractères—voir plus haut—sur la nature d'un fichier—voir partie (a) également—ou sur une variable numérique.

La syntaxe diffère dans ce dernier cas:

if (*operande1 operateur operande2*)

l'*operateur* étant plus explicite:

== != < <= > >=

On peut combiner des tests avec && et ¦¦ dont le sens est parfois inversé par rapport au langage C en raison d'un bug. La négation est introduite par !
On peut combiner des tests à l'aide de parenthèses sans problème.

Il est encore possible de remplacer des tests multiples par un <u>switch...case:</u>

switch (*variable*)
case *cas1*:	*commandes*
breaksw
...
case *casN*:	*commandes*
breaksw
default: *commandes*
breaksw
endsw

Chaque cas est terminé par breaksw et le switch lui même est terminé par endsw.
Les cas peuvent comporter des jokers * et ?

Les boucles

On dispose en C-shell de boucles while, repeat et foreach.

Les boucles while:
Leur syntaxe est en C-shell	while (*test*)
commandes
end

La boucle se répète toujours tant que le *test* est satisfait.

Les boucles repeat:
Leur syntaxe est	repeat *nb commande*

On répète alors une seule *commande* un certain *nb* de fois.

Les boucles foreach:
Elles sont semblables aux boucles for des autres shells et portent sur des listes:

foreach *variable* (*liste*)
commandes
end

break et continue:	Voir plus haut.

goto:

Le C-shell supporte les sauts avec goto vers un label placé en début de ligne
et suivi de deux points.

La récupération des paramètres transmis à un script C-shell

Les paramètres reçus sont ici désignés par $argv[0], $argv[1], . . . , $argv[N]

$argv[0] est le nom du script lui même.
Les autres sont les paramètres effectifs.
$#argv est le nombre de paramètres effectivement transmis.

Pour une manipulation globale signalons aussi
$argv qui est la chaîne de tous les paramètres et $argv[*] qui est la liste de tous les paramètres.

9) Sauvegarde des données — Rapports avec MS-DOS.

a) l'utilitaire tar:

Il est impérieux de sauvegarder ses données sur un support magnétique externe,
voire même de sauvegarder tout le système de fichiers, au cas où une anomalie
ou une erreur accidentelle surviendrait.

Pour cela UNIX dispose d'un utilitaire de sauvegarde à tout faire nommé tar
(tape archive) qui est capable de sauvegarder et restaurer des fichiers sur disquette,
cartouche 1/4 de pouce, dat ...

Syntaxe: tar -cvf /dev/*devicename files*
tar -xvf /dev/*devicename* [*files*]
tar -tvf /dev/*devicename* [*files*]

L'option v (verbose) impose à tar d'afficher à l'écran les fichiers traités.
L'option f stipule de travailler sur le périphérique *devicename* du répertoire /dev.

Les options principales retenues ici sont ensuite c, x et t dont les rôles sont:

Option c: (copy) Effectue une sauvegarde des fichiers spécifiés dans *files*.
files peur contenir des jokers * et ?
La sauvegarde est récursive, et comprend tous les sous répertoires du répertoire éventuellement mentionné dans *files*.
Par défaut tar travaille sur le répertoire courant.

Option x: (extract) Restaure les fichiers spécifiés dans *files* qui ne peut hélas contenir de jokers. Sans paramètre *files* tar restaure tous les fichiers de l'archive présente dans le périphérique *devicename*.

Option t: (tell) Liste à l'écran les fichiers sauvegardés sur l'archive. Le paramètre *files* permet de ne rechercher que certains fichiers, mais ne peut contenir de jokers.

tar permet de sauvegarder des Mégaoctets ou des Gigaoctets de données sans problème.
La sauvegarde peut se poursuivre sur plusieurs supports (disquettes ou cartouches) qui devront être soigneusement archivés et numérotés.

Les disquettes doivent en général être formatées avant usage avec la commande
format /dev/*devicename*

Le périphérique /dev/*devicename* peut prendre des formes très variables d'un système UNIX à un autre:

Exemples: Le lecteur de disquettes peut s'appeler

/dev/disquette	(SUN SOLARIS)
/dev/rfd0135ds18	(SCO UNIX)
/dev/fd0	(IBM AIX)

Le meilleur moyen de trouver le nom d'un périphérique est de consulter le fichier /etc/default/tar qui contient la liste des périphériques du système utilisables par tar.

Remarque: tar est très utile pour exporter des fichiers d'un système UNIX vers un autre dans la mesure où le format des archives tar est normalisé donc portable

d'un système à l'autre.

Exemple d'utilisation: (A consommer avec modération)

```
# pwd ↵
/home/maths/gilles
# tar -cvf /dev/fd0 * ↵ -- crée une archive sur disquette --
...
# tar -tvf /dev/fd0 ↵          -- vérifie l'archive constituée --
...
# rm -r * ↵                    -- supprime tout récursivement --
# tar -xvf /dev/fd0 ↵          -- restaure les fichiers perdus --
...
```

b) Les rapports avec MS-DOS:

De nombreux systèmes UNIX (IBM AIX, Systèmes sur PC...) permettent de manipuler des disquettes et des fichiers au format MS-DOS.

On trouve dans ce cas des commandes comme

dosformat /dev/*devicename*
dosdir ou dosls
dosread *source_dos dest_unix* ou doscp *source_dos dest_unix*
doswrite *source_unix dest_dos* ou doscp *source_unix dest_dos*

qui permettent de formater, lister une disquette MS-DOS, ou encore d'y lire ou écrire des fichiers.
Consulter le man pour en savoir plus ! (man dos par exemple).

Remarque: Il faut savoir que les fichiers texte (ascii) n'ont pas tout à fait le même format sous MS-DOS et sous UNIX. Dans le premier cas chaque ligne de texte est terminée par CR/LF, dans le second elle se termine seulement par LF.
En général les utilitaires de copie MS-DOS ↔ UNIX disposent d'options qui permettent de copier ces fichiers avec conversion des CR/LF en LF et vice versa.

Certains systèmes vont beaucoup plus loin en proposant une émulation MS-DOS qui permet de faire tourner des applications MS-DOS—pas trop sophistiquées— dans l'environnement UNIX.
C'est notamment le cas des systèmes UNIX sur PC qui permettent en plus d'accéder à la partition MS-DOS du PC s'il y en a une.

Enfin signalons que l'on trouve de plus en plus d'émulateurs MS-Windows permettant de faire fonctionner un certain nombre d'applications Windows en émulation dans une fenêtre X-Window.

10) Les terminaux texte.

Un terminal texte est composé d'un écran —24 ou 25 lignes et 80 colonnes en moyenne— et d'un clavier —très variable.

Dans une session de travail ordinaire le terminal est utilisé en **mode ligne**, c'est à dire que les données envoyées au terminal s'affichent ligne après ligne, les lignes qui montent étant peu à peu perdues.

Toutefois lors d'une utilisation plus fine du terminal on doit travailler en mode **pleine page**, par exemple avec un éditeur de texte comme vi ou avec une application pilotée par menus. Il faut alors gérer l'affichage avec un **curseur d'affichage** qui se déplace dans tout l'écran; il n'est alors plus question de scroller l'écran vers le haut, au risque de perdre des lignes !

Pour cela le terminal reconnaît un certain nombre de commandes qui permettent par exemple de:

- déplacer le curseur d'affichage
- effacer l'écran
- modifier les couleurs ou les attributs des caractères
- insérer ou supprimer une ligne sous le curseur
- insérer ou effacer un caractère sous le curseur
- etc.

Le curseur d'affichage est repéré par sa position à partir du coin supérieur gauche de l'écran.
Le terminal doit pouvoir distinguer les commandes des caractères à afficher dans le flot de données qui lui parvient. Aussi ces commandes commencent toujours par un caractère ESC (ascii 27) ou par un caractère de contrôle (ascii 0 à 26) et on les nomme donc séquences escape ou séquences de contrôle.

A la réception d'un tel caractère de début de commande, le terminal stocke les caractères suivants jusqu'à former une commande complète, qu'il exécute alors avant de revenir en mode normal.

Cela paraît assez simple, mais il y a en fait des complications:

- Les séquences de commande nécessitent parfois des paramètres —tels que la position du curseur s'il s'agit de le déplacer— qui peuvent prendre des formes très variables.

- Les séquences escape ne sont pas du tout standardisées, et chaque constructeur a adopté son propre style, variant même d'un terminal à l'autre d'une même marque !

Le pilotage d'un terminal par une application est donc assez délicat; tout d'abord le programme doit déterminer le modèle du terminal qui est connecté, ceci grâce à la variable d'environnement TERM ou term.
Mal renseignée, cette variable conduira à un comportement chaotique !

Ensuite le programme doit connaître les commandes acceptées par ce modèle de terminal; pour cela le système UNIX offre deux bases de données gigantesques nommées termcap (terminal capabilities) et terminfo (terminal information) situées respectivement dans /etc/termcap et /usr/lib/terminfo qui contiennent la description sous une forme particulière de toutes les commandes reconnues par des milliers de terminaux du commerce !

La base de données termcap est aujourd'hui obsolète et on utilise en fait terminfo.

L'administrateur peut ajouter à cette base de données la description d'un nouveau terminal si nécessaire, mais c'est encore une autre affaire.

Heureusement pour l'utilisateur final, il est rare que l'on doive accéder directement à la base de données terminfo.
En effet la plupart des systèmes proposent pour la programmation une bibliothèque de fonctions nommée **curses** qui permet de piloter les terminaux de façon à peu près transparente et avec un minimum de souffrances ! (cf § Programmation curses).

Ajoutons que la base de données terminfo décrit également les touches spéciales (touches de fonctions, flèches ...) de chaque terminal et les séquences escape ou de contrôle qu'elles génèrent.

Remarque: Le principe d'un émulateur de terminal sur PC, MAC ... est justement de générer et de reconnaître les mêmes séquences de commande que le terminal émulé.

Voyons un exemple de séquences générées (par le clavier) et reconnues (pour la gestion d'écran) par un terminal vt220 de DEC:

Exemple:

```
# Commandes reconnues en entrée par le terminal:

 am=    Off
 bel=   ^G        (Beep)            cr=     \r=  ^M  (Retour chariot)
 cub1=  \b=  ^H  (Back space)       ht=     \t=  ^I  (Tabulation)
 xon=   ^Q        (On line)         xoff=   ^S       (Off line)

 dim=   ESC[2m          (Sombre)
 smul=  ESC[4m          (Souligné)
 blink= ESC[5m          (Attribut clignotant)
 smso=  ESC[7m          (Inverse vidéo)
 sgr0=  ESC[m           (Annule tous les attributs)

 smacs=ESQ(0            (Active le mode semi-graphique)
 rmacs=ESC(B            (Désactive le mode semi-graphique)
```

```
# Commandes reconnues en entrée par le terminal (suite):

 clear=  ESC[2J=  ^L    (Efface l'écran)
 home=   ESC[H          (Curseur Home)
 sc=     ESC7           (Sauvegarde position du curseur)
 rc=     ESC8           (Restaure position du curseur sauvegardée)
 csr=    ESC[α;ßr       α et ß entiers (Définit les lignes de début
                        et de fin de la zone de scrolling)
 cup=    ESC[α;ßH       α et ß entiers  (Positionnement du curseur)
 cuu1=   ESC[A          (Déplacement du curseur d'une ligne vers le haut)
 cuu=    ESC[αA         α entier     (Déplacement de plusieurs lignes)
 cud1=   ESC[B          (Déplacement du curseur d'une ligne vers le bas)
 cud1=   ^J
 cud=    ESC[αB         α entier     (Déplacement de plusieurs lignes)
 cuf1=   ESC[C          (Déplacement du curseur d'une position vers la droite)
 cuf=    ESC[αC         α entier     (Déplacement de plusieurs positions)
 cub1=   ESC[D          (Déplacement du curseur d'une position vers la gauche)
 cub1=   ^H
 cub=    ESC[αD         α entier     (Déplacement de plusieurs positions)

 dch1=   ESC[P          (Suppression du caractère sous le curseur)
 ich1=   ESC[@          (Insertion d'un espace sous le curseur)
 dl1=    ESC[M          (Suppression de la ligne sous le curseur)
 il1=    ESC[L          (Insertion d'une ligne sous le curseur)
 el=     ESC[K          (Efface du curseur à la fin de  la ligne)
 ed=     ESC[J          (Efface du curseur à la fin de l'écran)

 ind=    ESCD=  ^J      (Scroll up de la zone de scrolling
                        si le curseur est sur la dernière ligne)
 ri=     ESCM           (Scroll down de la zone de scrolling)

# En mode semi-graphique les caractères sont interprétés de façon particulière:

 GV=   x    (Barre verticale)   GH=   q    (Barre horizontale)
 GC=   n    (Croisement)
 GL=   u    (Coté droit)        GR=   t    (Coté gauche)
 GU=   v    (Dessous)           GD=   w    (Dessus)
 G1=   k    (Coin)              G2=   l    (Coin)
 G3=   m    (Coin)              G4=   j    (Coin)
```

```
# Séquences émises par les touches spéciales du clavier:

kcuu1= ESC[A    (Flèche Haut)      kcud1= ESC[B    (Flèche Bas)
kcuf1= ESC[C    (Flèche Droite)    kcub1= ESC[D    (Flèche Gauche)

kbs=    ^H      (Back Space)       kbrk=  ^S       (Break)
kht=    ^I      (Tabulation)
kdch1= ^?       (Suppression)      kich1= ESC[2    (Insertion)

khome= ESC[H    (Home)
kpp=    ESC[5   (Page Up)          knp=   ESC[6    (Page Down)
khlp=   ESC[28~
kdo=    ESC[29~
kf1=    ESCOP   (Fonction 1)       kf2=   ESCOQ    (F2)
kf3=    ESCOR   (F3)               kf4=   ESCOS    (F4)
kf5=    ESC[16~ (F5)               kf6=   ESC[17~  (F6)
kf7=    ESC[18~ (F7)               kf8=   ESC[19~  (F8)
kf9=    ESC[20~ (F9)               kf10=  ESC[21~  (F10)
```

11) Les réseaux.

La plupart des systèmes UNIX fonctionnent aujourd'hui en réseau.
Le réseau lui-même est en général de type ethernet, sur lequel on utilise le protocole de communication TCP/IP qui est devenu un standard de fait sous UNIX.

a) Un aperçu de TCP/IP:

TCP/IP est un vaste ensemble de protocoles (Transmission Control Protocol et Internet Protocol) sur lequel repose l'ensemble du domaine internet à travers le monde.

Sans rentrer dans les détails, disons que chaque machine reliée à l'internet dispose de deux adresses:

- Une adresse physique, appelée adresse ethernet, codée sur 48 bits répartis en 6 chiffres hexadécimaux.
  Exemple: 00.12.E3.A0.28.13

  Cette adresse est en général codée de façon définitive dans le hardware de l'interface réseau de la machine.
  Chaque adresse est unique à travers le monde, les constructeurs se répartissant entre eux les plages d'adresses encore disponibles.

- Une adresse IP, codée sur 32 bits répartis en quatre chiffres décimaux.
  Exemple: 192.25.11.101

  Cette adresse IP est logicielle et est attribuée à chaque machine lors de l'installation du noyau TCP/IP.
  Toutefois cette adresse doit également être unique au monde, et des organismes centraux sont seul habilités à octroyer à une société donnée une plage d'adresses IP dans laquelle elle pourra puiser.
  En France la gestion est assurée par l'INRIA (Institut National de Recherche en Informatique et Automatique)
  On distingue plusieurs classes d'adresses IP, selon l'ampleur du sous réseau auquel elles appartiennent.

On n'utilise jamais directement les adresses ethernet, que seul le protocole manipule, et assez rarement les adresses IP qui sont assez difficiles à retenir.
Aussi pour les humains a-t-on mis en place un mécanisme de nommage qui permet de remplacer une adresse IP obscure par un mnémonique plus commode.
Pour cela le monde internet est divisé en domaines principaux correspondant en général à des pays, excepté pour les USA où l'on trouve plusieurs domaines principaux:

us	les USA en général
edu	organismes d'éducation des USA
com	organismes commerciaux des USA
gov	organismes gouvernementaux des USA

mil	organismes militaires des USA
uk	la Grande Bretagne
fr	la France

Dans chaque domaine, une société peut obtenir un sous domaine auprès d'un organisme central.

Exemple:	enst.fr	(L'ENST)
	u-cergy.fr	(L'université de Cergy)
	jussieu.fr	(L'université de Paris Jussieu)

On sépare le sous domaine du domaine par un point; on peut avoir plusieurs sous domaines imbriqués comme dans:

ccr.jussieu.fr (Centre de calcul et de recherche)

Enfin on termine par le nom de la machine elle-même, qui est déterminé par l'administrateur lors de l'installation.

Exemple: sun.u-cergy.fr
moka.ccr.jussieu.fr

b) Quelques services offerts par TCP/IP:

Les services TCP/IP que les utilisateurs d'UNIX sollicitent le plus sont:

- La connexion à une autre machine du réseau
- Le transfert de fichiers avec une autre machine du réseau
- Le courrier électronique

Le courrier électronique fait l'objet d'un chapitre particulier ci-après.
Pour ce qui est de la connexion à une autre machine, on dispose des services telnet et rlogin:

Syntaxe: telnet *machine.ssdomaine.domaine*
rlogin *machine.ssdomaine.domaine*

Ces deux services ouvrent une connexion réseau puis un shell sur la machine indiquée, permettant de travailler sur cette machine comme si on y était connecté directement.
On veillera à définir la variable TERM ou term en fonction du terminal de la machine locale si on utilise des applications en mode plein écran.

Tout cela suppose bien sûr d'avoir un compte utilisateur sur la machine indiquée, puisque telnet et rlogin exigent pour commencer un login et un mot de passe.
(rlogin suppose par défaut que le login est le même que sur la machine locale).

On met fin à la connexion comme d'habitude avec logout, exit ou ^D, ce qui nous ramène au shell de la machine locale.

Il peut arriver qu'une telle demande de connexion échoue car le nom de la machine est erroné.
Dans ce cas la commande ping permet de tester la validité du nom de machine:

Syntaxe: ping *machine.ssdomaine.domaine*

ping s'appelle ainsi car il envoie des paquets de données au destinataire qui se contente de les renvoyer tels quels en écho, afin de tester la liaison.

Si ping échoue—message du genre host unreachable—alors aucun service TCP/IP n'a de chance de fonctionner avec ce nom de machine.
Consulter l'administrateur si le problème persiste.

Le transfert de fichiers est du ressort de ftp qui tient son nom du protocole FTP (File Transfert Protocol):

Syntaxe: ftp *machine.ssdomaine.domaine*

ftp commence par exiger un login et un mot de passe sur la machine distante.
Si la connexion réussit, ftp n'ouvre pas de shell, mais ce place en mode commande avec un prompt ftp> souvent.

Pour obtenir la liste des commandes supportées faire help ↵ ou ? ↵
Citons les plus utiles d'entre elles:

bye Fin de la session ftp.
retour sur la machine locale.

pwd Affiche le répertoire courant sur la machine distante.
Il s'agit initialement du répertoire Home.

cd ou **chdir** Changement de répertoire sur la machine distante.

ls Liste des fichiers sur la machine distante.

get *source_file dest_file*

Transfert de fichier de la machine distante vers la machine locale.

put *source_file dest_file*

Transfert de fichier de la machine locale vers la machine distante.

Remarque Souvent les commandes get et put de ftp ne supportent pas les jokers.
Il faut alors utiliser les variantes **mget** et **mput** si elles sont disponibles.

De nombreux serveurs proposent des logiciels en shareware ou freeware que l'on peut récupérer par ftp; les intéressés n'ayant en général pas de compte sur le serveur en question, on utilise alors un ftp anonyme:

Le login est anonymous et le mot de passe associé est guest.
Bien sûr le compte correspondant dispose de possibilités restreintes pour des raisons de sécurité ! (Shell restreint, chroot ...).

Exemple de session ftp:

```
$ ftp sun.u-cergy.fr ↵
connected to sun.u-cergy.fr

User name: gilles ↵
Password for user gilles: -------- ↵

ftp> pwd ↵
/home/maths/gilles

ftp> ls ↵
prog1
prog2
prog1.c
prog2.c
projet_1995

ftp> get prog1.c prog1.c ↵
Transfering prog1.c ( 12562 Bytes )
Transfer complete

ftp> bye ↵
Connexion closed

$
```

Remarque: On peut très bien implémenter TCP/IP et les services telnet, ftp ...
sur un autre système qu'UNIX. Il existe des versions sous MS-DOS notamment.

12) Le e-mail.

Un des services de TCP/IP les plus utilisés à travers le monde internet est bien
le courrier électronique (e-mail).
Il permet d'envoyer un message à quiconque dispose d'un compte utilisateur sur une machine
reliée à l'internet.

Chaque machine gère le courrier entrant et sortant:

- le courrier sortant est expédié au destinataire par le réseau.
- le courrier entrant est trié et stocké dans une boite aux lettres (bal).

Il y a deux façons d'utiliser le gestionnaire de courrier mail, selon que l'on veut expédier
du courrier ou consulter le courrier reçu.

a) l'expédition de e-mail:

Pour désigner un utilisateur sur une machine donnée on fait précéder le nom de la machine
du nom de l'utilisateur, séparés par un caractère @.
D'où la syntaxe de la commande mail pour envoyer du courrier à *untel* sur la machine
machine.ssdomaine.domaine:

Syntaxe: mail *untel@machine.ssdomaine.domaine*

mail demande alors un entête subject qui doit permettre au destinataire de cerner rapidement
le contenu du message; cet entête d'une ligne en général se termine par la frappe de ↵

Commence alors le champ correspondance, dans lequel on peut taper le texte voulu,
en évitant toutefois les caractères spéciaux et les lettres accentuées qui sont parfois
mal interprétés d'une machine à une autre !

On met fin au champ correspondance en tapant ^D ou .↵ au tout début d'une ligne.
S'affiche alors parfois le prompt cc: (carbon copies) qui offre de dupliquer le message
vers d'autres correspondants dont on donne ici la liste.
A défaut, faire ↵ pour envoyer le mail.

Remarque: Un message peut ne pas arriver au destinataire si celui-ci est inconnu

de la machine destination; dans ce cas le démon de gestion du courrier de cette machine vous enverra un message d'erreur par le mail.

Le mode d'édition de base du courrier est médiocre dans la mesure où le terminal est exploité en mode lignes, ce qui ne permet pas de revenir sur une ligne déjà saisie !
Pour éditer le champ correspondance avec l'éditeur vi, il suffit au début de la première ligne de ce champ de taper ~v.
A la fin de la saisie, quitter vi en enregistrant le message comme d'habitude pour revenir dans mail.

Cela permet aussi d'insérer dans le mail tout ou partie d'un fichier existant qui sera joint à la correspondance.

Pour abandonner l'envoi d'un message, presser deux fois de suite la touche Break (intr = ^C)

b) La consultation du e-mail:

Il faut savoir que le courrier qui arrive est stocké dans une boite aux lettres centrale du répertoire /usr/spool/mail.
Le mail reste dans cette boite aux lettres jusqu'à ce qu'il soit consulté, après quoi il est déplacé dans la boite aux lettres locale Home/mbox.

Syntaxe: mail
mail -f mbox

mail tout court donne accès aux nouvelles fraîches qui n'ont pas encore été consultées.
Avec l'option -f mbox on peut relire du courrier déjà consulté ayant été archivé dans la mailbox locale.

Dans les deux cas, mail affiche la liste des entêtes de messages, précédés d'un numéro d'ordre, et se place en attente d'une commande avec le prompt &.
Le message courant est marqué par un > dans la liste.

commandes de consultation du mail (suivies de ¿)

h+	Afficher page suivante de la liste des entêtes
h-	Afficher page précédente de la liste des entêtes
h	Réafficher page courante de la liste des entêtes
h*	Lister successivement tous les entêtes de messages
h*username*	Lister tous les entêtes de messages provenant de telle personne
h/*texte*	Lister tous les entêtes de messages comportant le *texte* indiqué
nb	Consulter le message de numéro *nb* dans la liste

e*nb*	Editer le message numéro *nb* dans l'éditeur stipulé par la variable d'environnement EDITOR ou editor Sans *nb* on édite le message courant.
s*nb filename*	Sauver le message complet de numéro *nb* dans le fichier *filename* Sans l'option *nb* on sauve le message courant
w*nb filename*	Sauver le champ correspondance du message de numéro *nb* dans le fichier *filename* Sans l'option *nb* on sauve le message courant
r*nb*	Répondre immédiatement à l'expéditeur du message numéro *nb* Voir mode expédition à ce sujet Sans option *nb* il s'agit de répondre au message courant
d*nb*	Supprimer le message de numéro *nb* Sans option *nb* supprime le message courant
u*nb*	Restaurer le message de numéro *nb détruit avec d*nb Sans option *nb* restaure le message courant
F*nb untel*	Faire suivre le message de numéro *nb* vers *untel* Sans option *nb* fait suivre le message courant
q	Quitter mail et enregistrer les modifications
ex	Quitter mail sans enregistrer les modifications

Un utilisateur peut de temps en temps détruire le fichier Home/mbox afin de faire de la place, lorsque le courrier qu'il contient n'a plus d'intérêt.

Signalons qu'il est possible de faire suivre son courrier sur une autre machine reliée à l'internet en cas d'absence prolongée.
Il suffit pour cela de créer dans son répertoire Home un fichier nommé .forward contenant la nouvelle adresse électronique à laquelle le mail doit être réexpédié, sous la forme *untel@machine.ssdomaine.domaine*

Exemples:

```
$ mail leon@ibm.u-cergy.fr
Subject: Demande de numero de fax ↵

Pouvez vous s'il vous plait me faire parvenir par retour du mail
votre numero de fax pour envoi du document demande.
Merci
Je reste a votre disposition pour tout renseignement.
.↵

cc: ↵
Mail sent to user leon@ibm.u-cergy.fr
$
```

```
$ mail
/usr/spool/mail/gilles: 2 messages
>1 N    dupont  Sep 15 11:45   "Probleme de modem"
  2 N    durand  Sep 15 12:02   "Reunion de departement"
&2 ↵
Mail from dupont@sun.u-cergy.fr        Sep 15 11:45
Reunion de departement

Une reunion de departement est prevue le 20 septembre
a 16h00 salle 208 Bat A.
Comptons sur la presence de tous.
Merci.
&q ↵
Held 1 message in /usr/spool/mail/gilles
Saved 1 message in /home/maths/gilles/mbox
$
```

II) Fonctions de manipulation de fichiers et répertoires.

1) Opérations sur les fichiers:

Un fichier est un ensemble de données (octets) stockées sur disque dur, disquette, CDROM...
Les données peuvent être organisées en lignes de texte (fichier texte) ou purement binaires (fichier binaire).

Les opérations de base sur un fichier sont:

- création ou ouverture
- déplacement dans le fichier
- lecture, écriture
- fermeture

Sous UNIX un fichier est identifié par un nom de 256 lettres au maximum.
En plus de ce nom importe le nom du répertoire dans lequel est situé le fichier.

Exemples: toto.txt
/usr/bin/whodunit

Sont des noms de fichiers valides.

Il existe en C deux bibliothèques d'accès aux fichiers:

- Une bibliothèque de bas niveau nécessitant les headers <unistd.h> et <fcntl.h>.
- Une bibliothèque de haut niveau nécessitant <stdio.h>.

Les fonctions de haut niveau permettent des opérations plus sophistiquées que les autres.
Elles s'appuient en fait—sans qu'on le sache—sur les fonctions de bas niveau.
Les fonctions de bas niveau sont appelées ainsi car elles sont plus proches du système d'exploitation, et exploitent directement les données sur disque tandis que les fonctions de haut
niveau utilisent des tampons par lesquels transitent les données lors de la lecture
ou de l'écriture.

L'intérêt de ces tampons est de limiter le nombre d'accès au disque: en cas de lecture ou écriture de plusieurs petits paquets de données, ceux ci sont regroupés en un seul paquet dans le tampon et c'est le tampon qui fera l'objet d'une seule opération sur disque.

a) Les fonctions de bas niveau:

Les fonctions de bas niveau manipulent un fichier à l'aide d'un handle (descripteur)
qui est associé au fichier lors de l'ouverture ou de la création et qui servira à tous les accès ultérieurs jusqu'à la fermeture du fichier.
Le nom du fichier ne sert qu'à obtenir ce handle de la part du système d'exploitation.

Ouverture et fermeture d'un fichier:

L'ouverture d'un fichier passe par open( ) ou creat( ) et sa fermeture par close( ).

Syntaxe:

```
int open( char *nom_fichier, int omode, [int amode]);
int creat( char *nom_fichier, int amode );
int close( int handle );
```

La fonction open( ) reçoit une chaîne contenant le nom du fichier—avec éventuellement un chemin d'accès.
Elle reçoit aussi le mode d'ouverture voulu par le biais de constantes prédéfinies que l'on peut combiner par ¦ dans omode:

O_RDONLY, O_WRONLY, O_RDWR,
Selon que l'accès se fera en lecture, écriture ou les deux.

O_TRUNC	Pour vider le fichier (dangereux !)
O_CREAT	Pour en imposer la création s'il n'existe pas.
O_APPEND	Pour se positionner directement à la fin du fichier après l'ouverture.
O_NDELAY	Modifie le mode de lecture qui devient non bloquant.

Le paramètre amode est facultatif dans la fonction open( ):
on dispose des mêmes constantes que pour chmod( ) —voir plus loin— pour déterminer les droits du fichier à la création. Ces droits sont modulés par le masque de création de fichier —voir umask( ) ci-après.
On peut aussi y introduire les droits voulus directement en octal.

La fonction open( ) retourne en échange de tout cela un entier baptisé handle (descripteur) délivré par le système d'exploitation et qui servira lors des accès ultérieurs à ce fichier.

creat( ) est un raccourci parfois utilisé pour la création d'un fichier, avec remise à zéro de celui-ci s'il existe déjà.

```
creat( char *nom_fichier, amode );
équivaut à
open(char *nom_fichier, O_WRONLY ¦ O_CREAT ¦ O_TRUNC, amode );
```

La fonction creat( ) renvoie de même un handle pour manipuler le fichier.

Remarques: En cas d'erreur open( ) et creat( ) renvoient -1. Il est préférable de vérifier cette valeur de retour afin de ne pas continuer des opérations sur un fichier inexistant !

La fonction close( ) reçoit un handle. Elle ferme le fichier associé à ce handle par open( ) ou creat( ). Elle renvoie -1 en cas d'erreur.

Les drivers de périphériques peuvent être gérés comme des fichiers, sauf qu'il n'est pas toujours nécessaire de les ouvrir et de les fermer.
Ainsi l'entrée standard, la sortie standard et la sortie erreur standard sont ouverts en permanence et ont les handles respectifs 0, 1 et 2.
Exemple:

```
int hd1, hd2;
...
hd1= creat( "toto.txt", 0777 );
hd2= open( "/bin/filexyz", O_RDWR ¦ O_APPEND );
...
close( hd1 );
close( hd2 );
```

Déplacement dans un fichier:

On peut se déplacer dans un fichier ouvert grâce à un curseur (je n'ose pas dire pointeur !) de lecture / écriture qui indique l'endroit où se fera la prochaine opération, par rapport au début du fichier.
Le curseur est par défaut placé au début du fichier lors de l'ouverture, sauf dans le mode d'ouverture O_APPEND où il est placé à la fin.

Une opération d'écriture ne peut se faire qu'à la fin d'un fichier, mais la lecture peut avoir lieu n'importe où dans le fichier, en déplaçant ce curseur.

Une fonction est disponible pour cela, lseek( ):

Syntaxe: long lseek(int handle, long dep, int org);

La fonction lseek( ) reçoit bien sûr le handle du fichier concerné, mais surtout un long int qui détermine de combien d'octets le curseur sera déplacé, et une constante org qui indique à partir de quelle origine se fait le déplacement:

SEEK_SET déplacement absolu en avant depuis le début du fichier.
SEEK_CUR déplacement relatif en avant par rapport à la position courante.
SEEK_END déplacement absolu en arrière par rapport à la fin du fichier.

Lecture et écriture dans un fichier:

Les incontournables fonctions read( ) et write( ) sont là pour cela:

Syntaxe: int read(int handle, char *zone, unsigned nb);

int write(int handle, char *zone, unsigned nb);

La lecture avec read( ) a lieu à la position courante du curseur, lequel est déplacé vers l'avant
du nombre d'octets lus.
L'écriture ne peut—pour des raisons techniques—avoir lieu qu'à la fin du fichier.

En plus du handle incontournable ces fonctions reçoivent l'adresse d'une zone
(un tableau de caractères en général) où les octets à transférer seront lus on écrits.

Le nombre d'octets à transférer est donné par l'entier nb.
La fonction renvoie le nombre d'octets effectivement lus ou écrits, ou -1 en cas d'erreur.
Remarque: Lors de la lecture dans un fichier spécial de type driver de périphérique, ou tube fifo la lecture peut être bloquante ou non bloquante selon que le flag O_NDELAY a été levé ou non lors de l'appel à open():

O_NDELAY non levé	En cas de lecture, s'il n'y a pas de caractère disponible read( ) reste en attente jusqu'à ce qu'il y en ait.
O_NDELAY levé	En cas de lecture, s'il n'y a pas de caractère disponible read( ) termine immédiatement avec un code de retour nul.

Voir également l'option F_SETFL de la primitive fcntl( ) pour modifier ce flag.

Exemple:

```
int hd, len;
char texte[80], copie[80];
...
hd= open( "essai.txt", O_CREAT );     /* création */
if( hd == -1 ) printf( "erreur d'ouverture\n" );
else
        {
        strcpy( texte, "ceci est un essai \n" );
        len= strlen( texte );
        write( hd, texte, len );          /* écriture texte */
        lseek( hd, 0, SEEK_SET );         /* retour au début du fichier */
        read( hd, copie, len ),           /* relit le fichier */
        printf( copie );                  /* devrait afficher le message texte */
        close( hd ),                      /* fermeture indispensable */
        }
```

Caractéristiques d'un fichier:

La fonction fstat( ) permet d'accéder aux caractéristiques d'un fichier ouvert:

Syntaxe: int fstat(int handle, struct stat *buf_ptr);

Elle nécessite l'inclusion des headers <sys/stat.h> et <sys/types.h>

La fonction fstat( ) reçoit l'adresse d'une structure de type prédéfini stat déclarée dans le programme et dont les champs principaux sont:

```
struct stat
        {
        ...
        unsigned st_mode;           /* type et droits d'accès */
        unsigned st_uid;            /* identification du propriétaire */
        unsigned st_gid;            /* identification du groupe d'appartenance */
        unsigned long st_size;      /* taille du fichier */
        unsigned st_mtime;          /* date de dernière modification */
        }
```

Au retour la fonction fstat( ) retourne -1 en cas d'erreur (handle non valide ...) ou 0 en cas de succès, auquel cas la structure stat pointée par buf_ptr est complétée.

Remarque: La fonction stat( ) que l'on verra plus loin fait de même à partir du nom d'un fichier, qui n'est pas nécessairement ouvert.

Le champ st_mode contient le type du fichier et les droits d'accès, que l'on peut tester à l'aide de constantes prédéfinies.
Pour le type on dispose des constantes suivantes:

S_IFREG	Fichier régulier
S_IFDIR	Fichier répertoire
S_IFLNK	Lien symbolique
S_IFBLK	Driver de périphérique bloc
S_IFCHR	Driver de périphérique caractère
S_IFIFO	Tube nommé

On testera en fait le type à l'aide des macro-fonctions suivantes:

```
S_ISDIR( unsigned mod );
S_ISREG( unsigned mod );
S_ISLNK( unsigned mod );
S_ISBLK( unsigned mod );
S_ISCHR( unsigned mod );
S_ISFIFO( unsigned mod );
```

Il suffit de passer st_mode à une de ces fonctions pour qu’elle réponde TRUE ou FALSE selon que le fichier est du type indiqué ou non.

Pour ce qui est des droits d'accès on dispose des mêmes constantes que pour chmod( ) (Voir plus loin).

b) Les fonctions de haut niveau:

Les fonctions de haut niveau manipulent cette fois un fichier à l'aide d'un pointeur sur une structure de type prédéfini FILE. Ce pointeur servira à tous les accès ultérieurs jusqu'à la fermeture du fichier.

Le nom du fichier ne sert qu'à compléter la structure FILE associée et à obtenir le pointeur sur cette structure.
Il n'est pas nécessaire de connaître le contenu de la structure FILE pour utiliser ces fonctions !
Elles nécessitent les headers <stdio.h> et parfois <sys/stat.h>

Ouverture et fermeture d'un fichier:

Deux fonctions remplissent ces rôles, fopen( ) et fclose( ).

Syntaxe:

```
FILE * fopen( char *nom_fichier, char *fmode );
int fclose( FILE *f_ptr );
```

La fonction d'ouverture fopen( ) reçoit une chaîne de caractères représentant le nom du fichier,
avec si nécessaire un chemin d'accès.
On précise le mode d'ouverture du fichier à l'aide d'une chaîne de caractères fmode:

"r"	accès en lecture
"w"	accès en lecture et écriture
"a"	accès en ajout c'est à dire écriture à la fin du fichier
"w+"	création d'un nouveau fichier en lecture et écriture
"a+"	création d'un nouveau fichier en ajout
"b"	ouverture en mode Binaire

En fait fmode est une concaténation de la forme "r" ou "w+b"...
Par défaut le fichier est ouvert en mode Texte.

fopen( ) retourne un pointeur sur une structure de type FILE, qui est gérée par fopen( ).
On ne déclarera pas cette structure sans le programme mais seulement un pointeur dessus, qui recevra le résultat de fopen( ).
Si le pointeur est NULL, l'ouverture ou la création a échoué. On vérifiera cette valeur de retour pour éviter des opérations ultérieures sur un fichier fantôme.

La fonction fclose( ) ferme le fichier associé au pointeur f_ptr transmis par fopen( ).
Elle assure avant la fermeture la synchronisation des données, c'est à dire l'écriture des données encore en attente dans le tampon d'écriture.

Exemple:

```
FILE *f_ptr;
...
f_ptr = fopen( "./toto.txt", "r" );
...
fclose( f_ptr );
```

Déplacement dans un fichier:

Comme pour les fonctions de bas niveau, un fichier ouvert dispose—dans la structure FILE associée—d'un curseur (j'hésite toujours à dire un pointeur) qui indique l'endroit dans le fichier où auront lieu les prochaines opérations de lecture/écriture.
A l'ouverture le curseur est au début du fichier, sauf en mode ajout.

L'écriture ne peut avoir lieu qu'à la fin du fichier, mais la lecture peut avoir lieu en tout point du fichier au gré du déplacement du curseur.

Les fonctions associées sont fseek( ), ftell( ) et rewind( ).

Syntaxe:

```
int fseek( FILE *f_ptr, long dep, int org );
long ftell( FILE *f_ptr );
void rewind( FILE *f_ptr );
```

La fonction fseek( ) sert à déplacer le curseur dans le fichier du nombre d'octets dep, par rapport à l'origine org conformément aux constantes:

SEEK_SET	déplacement par rapport au début du fichier
SEEK_CUR	déplacement par rapport à la position actuelle
SEEK_END	déplacement en arrière par rapport à la fin du fichier

Elle renvoie -1 en cas d'erreur.

Inversement la fonction ftell( ) renvoie la position actuelle du curseur par rapport au début du fichier associé à f_ptr.

La fonction rewind( ) ramène le curseur au début du fichier associé à f_ptr;
rewind(f_ptr) équivaut à fseek(f_ptr, 0, SEEK_SET);

Lecture et écriture dans un fichier:

La bibliothèque de haut niveau offre davantage de possibilités au niveau des fonctions de lecture / écriture que la bibliothèque de bas niveau: on peut lire ou écrire des caractères, chaînes, variables, blocs de données avec diverses fonctions:

Syntaxe:

```
fputc( int chr, FILE *f_ptr );
int fgetc( FILE *f_ptr );
```

```
fputs( char *chaine, FILE *f_ptr );
char * fgets( char *chaine, int nb, FILE *f_ptr );

fprintf( FILE *f_ptr, char *format, var1, ..., varN );
fscanf( FILE *f_ptr, char *format, &var1, ..., &varN );

fwrite( void *zone, unsigned taille, unsigned nb, FILE *f_ptr );
unsigned fread( void *zone, unsigned taille, unsigned nb, FILE *f_ptr );
```

Les fonctions fputc( ) et fgetc( ) écrivent et lisent caractère par caractère dans un fichier.
Les int de la syntaxe seront comme d'habitude copieusement remplacés par des char.

Les fonctions fputs( ) et fgets( ) permettent d'écrire et lire une chaîne de caractères dans un fichier.
La lecture par fgets( ) s'arrête lorsqu'un caractère newline (\n) est rencontré, lorsque la fin du fichier est rencontrée, ou lorsque le nombre maximal nb de caractères a été lu.
Dans ce premier cas le caractère \n est écrit avec la chaîne, qui est de plus terminée par un caractère NUL. fgets( ) retourne un pointeur NULL en fin de fichier.

Les fonctions fprintf( ) et fscanf( ) fonctionnent comme printf( ) et scanf( ), mais elle renvoient leurs résultats ou prennent leurs données dans le fichier indiqué.
En général un fichier de données écrit de façon structurée avec des fprintf( ) sera relu avec des
fscanf( ).

Enfin les fonctions fwrite( ) et fread( ) permettent d'écrire des blocs de données—tableaux, structures—dans un fichier.
En plus de l'adresse de la zone mémoire contenant les données à écrire ou devant recevoir les donnés lues, on indique la taille de chaque bloc de données, et le nombre de blocs à écrire
ou lire.
Le volume des données manipulées est donc égal à taille * nb octets.

<u>Exemple:</u>

```
FILE *f_ptr;
int len;
char texte[80], copie[80];
...
f_ptr = fopen( "toto.txt", "w+" );
if( f_ptr = = NULL ) printf( "erreur d'ouverture de fichier\n" );
else
        {
        strcpy( texte, "second essai\n" );
        len= strlen( texte );
        fputs( texte, f_ptr );
        fseek( f_ptr, 0, SEEK.SET );
        fgets( copie, len, f_ptr );
        printf( copie );
        fclose( f_ptr );
        }
```

Synchronisation des données:

La fonction fflush( ) force la synchronisation des données d'un fichier, c'est à dire qu'elle force
l'écriture dans le fichier du contenu du tampon d'écriture, et qu'elle vide le tampon de lecture.
L'appel à cette fonction n'est pas nécessaire car la synchronisation est effectuée périodiquement. On ne l'utilisera que dans des cas particuliers, pour réinitialiser les tampons.

Syntaxe: `fflush( FILE *f_ptr );`

Exemple: `fflush( stdin );`

Il faut savoir que stdin et stdout sont des pointeurs prédéfinis sur les drivers
de périphériques associés à l'entrée et à la sortie standard, c'est à dire le clavier et l'écran.
L'opération fflush(stdin) a pour but de vider le tampon du clavier, par où transitent
les caractères frappés encore en attente de traitement

2) Les fonctions de manipulation de fichiers:

En plus des fonctions précédentes permettant des opérations sur le contenu des fichiers,
il existe des fonctions manipulant les fichiers eux même, afin de les renommer, supprimer,
d'en changer les droits d'accès.
Elles nécessitent l'inclusion des headers <unistd.h> et parfois <sys/stat.h>.
Syntaxe: `int rename( char *old_name, char *new_name );`

```
int remove( char *nom_fichier );
int unlink( char *nom_fichier );
int symlink( char *nom_fichier, char *nom_lien );
int link( char *nom_fichier, char *nom_lien );

int chmod( char *nom_fichier, int amode );
int stat( char *nom_fichier, struct stat *buf_ptr );
int umask( int masque );
```

La fonction rename( ) renomme le fichier indiqué par old_name et lui donne le nom new_name.
Ces deux noms peuvent contenir un chemin.
Si les deux chemins diffèrent, le fichier est déplacé dans le nouveau répertoire indiqué.

La fonction remove( ) supprime le fichier indiqué
En fait remove( ) fait appel à la fonction unlink( ) dont le rôle consiste à détruire un lien, si bien que lorsque le fichier visé n'est qu'un lien, ce lien est supprimé sans pour autant que le fichier original ne soit affecté !

La fonction symlink( ) crée comme son nom l'indique un lien symbolique vers un fichier ou un répertoire existant dans l'arborescence.
Par ailleurs link( ) crée un lien physique qui ne peut concerner qu'un fichier.

La fonction chmod( ) modifie les droits d'accès au fichier.
Les nouveaux droits d'accès au fichier sont déterminés par les constantes prédéfinies:

S_IRWXU	Droits rwx pour le propriétaire
S_IRWXG	Droits rwx pour le groupe d'appartenance
S_IRWXO	Droits rwx pour les autres utilisateurs
S_IRUSR	Droit r pour le propriétaire
S_IWUSR	Droit w pour le propriétaire
S_IXUSR	Droit x pour le propriétaire
S_IRGRP	Droit r pour le groupe d'appartenance
S_IWGRP	Droit w pour le groupe d'appartenance
S_IXGRP	Droit x pour le groupe d'appartenance
S_IROTH	Droit r pour les autres utilisateurs
S_IWOTH	Droit w pour les autres utilisateurs
S_IXOTH	Droit x pour les autres utilisateurs
S_IAMB	Droits rwx pour tous
S_ISUID	Droit su pour un exécutable
S_ISGID	Droit sg pour un exécutable

S_ISVTX	Sticky bit pour un exécutable ou un répertoire

Remarque: On peut aussi coder directement les droits en octal dans chmod( ).

La fonction stat( ) fonctionne comme fstat( ), mais à partir du nom du fichier.
(Voir plus haut).

Toutes ces fonctions retournent -1 en cas d'erreur, notamment si le fichier n'existe pas.

Signalons enfin la fonction umask( ) qui positionne le masque de création des fichiers et répertoires. En effet lors de la création d'un fichier ou d'un répertoire, certains droits sont demandés. Ils sont en fait modulés par la valeur masque, qui permet d'inhiber systématiquement certains droits.
Le masque en question est propre au processus en cours.
La fonction retourne l'ancien masque de création des fichiers.

3) Les fonctions de manipulation de répertoires:

Lorsqu'on manipule des fichiers on peut être amené à se déplacer dans l'arborescence des répertoires du système.
On peut aussi créer ou supprimer un répertoire.
De nombreuses fonctions sont disponibles pour réaliser ces opérations.

a) Opérations sur les répertoires:

Les répertoires sont gérés par les fonctions getcwd( ), chdir( ), mkdir( ), rmdir( ):

Syntaxe:

```
char * getcwd( char *chemin, int len );
int chdir( char *chemin );
int mkdir( char *chemin, int amode );
int rmdir(char *chemin );
int rename( char *old_name, char *new_name );
```

getcwd( ) récupère dans la chaîne chemin le répertoire courant du processus.
Le paramètre len est un nombre maximum de caractères que la fonction pourra stocker dans chemin.
Sous UNIX on pourra opter pour len = 256 car un chemin dépasse rarement cette longueur.
Une constante est parfois prédéfinie nommée MAXPATHLEN ou MAXPATHSIZE.
getcwd( ) retourne un pointeur sur la chaîne chemin.

chdir( ) permet de changer de répertoire. On peut utiliser un déplacement absolu par rapport
à la racine / , ou un déplacement relatif au répertoire actuel, y compris à l'aide de . (répertoire actuel) et .. (répertoire au dessus).

En cas d'erreur chdir( ) retourne -1, sinon elle retourne 0.

Ensuite mkdir( ) et rmdir( ) créent ou suppriment le répertoire indiqué.
mkdir( ) reçoit les droits d'accès attribués au répertoire nouvellement crée, modulés par le masque de création de fichier défini par umask( ).

Elles retournent aussi -1 en cas d'impossibilité et sinon 0.
Remarques: La fonction rmdir( ) ne supprime pas forcément le répertoire indiqué. S'il s'agit d'un lien vers un autre répertoire, c'est le lien qui est supprimé, sans que l'original ne soit affecté.
Une tentative de création de répertoire par mkdir( ) à travers un lien échouera systématiquement.

Par contre les liens symboliques sont traversés par chdir( ).
Il convient donc de prendre des précautions lors du parcours récursif d'une arborescence: on ne doit pas traverser les liens symboliques sans quoi on risque de créer une boucle infinie.
Dans le même esprit attention à chdir("..") qui ne ramène pas toujours là où l'on croit après la traversée d'un lien symbolique !

La fonction rename( ) vue pour les fichiers peut également servir à renommer un répertoire. Le répertoire ne peut toutefois pas être déplacé.

Exemple:

```
char rep[256];

getcwd( rep, 256 );
mkdir( "toto", 0777 );          /* crée toto */
chdir( "toto" );                /* passe dans toto */
mkdir( "truc", 0777 );          /* crée toto/truc */
chdir( rep );                   /* retour dans le répertoire d'origine */
rmdir( "toto/truc" );
rmdir( "toto" );        /* supprime ces deux répertoires */
```

b) Recherche d'un fichier dans un répertoire:

Pour accéder à la liste des fichiers d'un répertoire, on doit ouvrir ce répertoire avec une fonction spécifique nommée opendir( ), après quoi on peut consulter la liste des fichiers à l'aide de readdir( ), puis on referme le répertoire avec closedir( ).
Ceci parce qu'un répertoire est structuré d'une façon particulière et ne peut guère être manipulé
avec les fonctions usuelles.

Ces fonctions nécessitent l'inclusion de <dirent.h> ou <sys/dirent.h>

Syntaxe:
```
DIR * opendir( char *dirname );
struct dirent * readdir( DIR * dir_ptr );
closedir( DIR * dir_ptr );
```

La fonction opendir( ) retourne un pointeur sur une structure de type prédéfini DIR.
On ne déclarera dans le programme qu'un pointeur sur cette structure, qui servira à toutes les manipulations ultérieures.

En cas d'échec (répertoire inexistant ...) la fonction retourne le pointeur NULL.

Ensuite on passe à readdir( ) le pointeur précédemment obtenu, en échange de quoi la fonction
retourne un pointeur sur une autre structure prédéfinie nommée dirent, contenant divers renseignement sur le prochain fichier du répertoire:

```
struct dirent
    {
    ...
    char d_name[ ]
    }
```

Le champ d_name est le plus important; il contient le nom de l'entrée trouvée.
Un appel à stat( ) permettra alors d'en savoir plus sur cette entrée.

Les appels successifs à readdir( ) passent en revue toutes les entrées du répertoire, quelque soient leur type (fichier régulier, sous répertoire...).
La structure interne utilisée est écrasée à chaque appel !
Lorsque la liste est épuisée readdir( ) renvoie un pointeur NULL.

Enfin closedir( ) referme le répertoire ouvert par opendir( ).

III) Gestion des processus.

Rappelons qu'un processus est un programme en cours d'exécution.
Dans l'environnement multitâche d'UNIX, de nombreux processus sont en cours
à un instant donné.

Un processus peut faire appel à d'autres processus, de deux façons différentes:

- Le processus appelé remplace le processus appelant
  et s'exécute à sa place.

- Le processus appelé s'exécute parallèlement au processus appelant.

On peut programmer en C l'appel d'un processus père par un processus fils à l'aide
des fonctions exec...( ) et fork( ).

1) L'appel à exec...( ):

Il existe plusieurs variantes d'exec...( ) selon la forme des paramètres à passer
et l'environnement voulu.

Syntaxe:

```
int execl( char *progname, char *arg0, ..., char *argN, NULL );
int execle( char *progname, char *arg0, ..., char *argN, NULL, char **env );
int execlp( char *progname, char *arg0, ..., char *argN, NULL );
int execlpe( char *progname, char *arg0, ..., char *argN, NULL, char **env );
int execv( char *progname, char **argv );
int execve( char *progname, char **argv, char **env );
int execvp( char *progname, char **argv );
int execvpe( char *progname, char **argv, char **env );
```

Ne nous laissons pas abuser par la multitude de variantes d'exec...( ) !

Le principe des fonction exec...( ) est le suivant:
Le programme désigné par progname, avec éventuellement un chemin d'accès, devient
le processus actif, à la place du processus père qui se termine avec l'appel à exec...( ).

Il est donc vital de comprendre qu'il n'y a pas de retour d'un exec...( ) réussi puisque
le processus appelant est démantelé.
Il n'y a retour qu'en cas d'erreur (programme inexistant ...) avec la valeur -1.
On devra envisager cette éventualité.

Les fonctions execl...( ) reçoivent en plus un nombre déterminé d'arguments arg0, ..., argN
qui sont les chaînes de caractères à passer en paramètre au processus fils.

La liste de ces arguments se termine par un NULL. Par convention l'argument arg0 est toujours présent et est le nom du processus fils. (cela permettra au processus fils de savoir comment il s'appelle).

Exemple:

```
int e;
...
e = execl( "/usr/bin/cp", "cp", "toto.c", "toto.bak", NULL );
            /* lance cp toto.c toto.bak */
if ( e = = -1 ) printf( "Erreur d'exécution\n" );
```

Les fonctions execv...( ) reçoivent les paramètres d'une façon différente, sous la forme d'un tableau de pointeurs sur des chaînes de caractères char *argv[].
Le dernier élément de ce tableau est un NULL marquant la fin de la liste.
On utilise les variantes execv...( ) lorsque le nombre de paramètres transmis n'est pas connu d'avance.

Exemple:

```
int e;
char *argv[ ] = { "cp", "toto.c", "toto.bak", NULL };
...
e = execv( "/usr/bin/cp", argv );
            /* même résultat que ci-dessus */
if ( e = = -1 ) printf( "Erreur d'exécution\n" );
```

Les variantes execlp( ) et execvp( ) opèrent comme execl( ) et execv( ) sauf qu'elles recherchent le programme progname dans le PATH (ou path) si nécessaire.

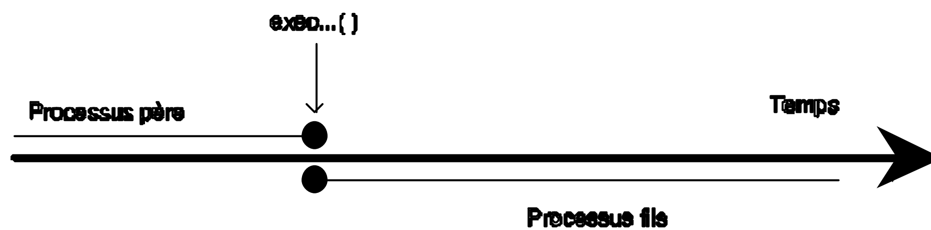
Les variantes execle( ) et execve( ) reçoivent un paramètre d'environnement supplémentaire: il s'agit d'un tableau de pointeurs sur des chaînes de caractères, terminé par un NULL, représentant des variables d'environnement à ajouter à l'environnement du processus fils. Par défaut le processus fils reçoit une copie de l'environnement de son père.

Enfin execlpe( ) et execlve( ) combinent les avantages de deux variantes précédentes.

Exemple:

```
int e;
char *env[ ] = { "INSTALL=/usr/local/bin", NULL };
...
e = execle( "sonny", "sonny", NULL, env );
            /* le père transmet une variable d'environnement à son fils */
if ( e = = -1 ) printf( "Erreur d'exécution\n" );
```

Principe d'exec...( ):



Le processus fils hérite de nombreuses propriétés de son père lors de l'exec...( ).
Voir le chapitre consacré aux processus à ce sujet.

Exemple d'exec...( ):

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void main( )
        {
        int e;

        e= execl( "sonny", "sonny", NULL );
        if ( e == -1 )
                printf( "Erreur lors de l'exec.\n" );

        /* sinon il n'y a pas de retour */
        }
```

2) L'appel à fork( ):

Pour que le processus père continue son exécution parallèlement au processus fils nouvellement crée il est nécessaire de faire appel à fork( ).

Syntaxe: int fork(void);

L'appel à fork( ) effectue une scission du processus en deux processus identiques:

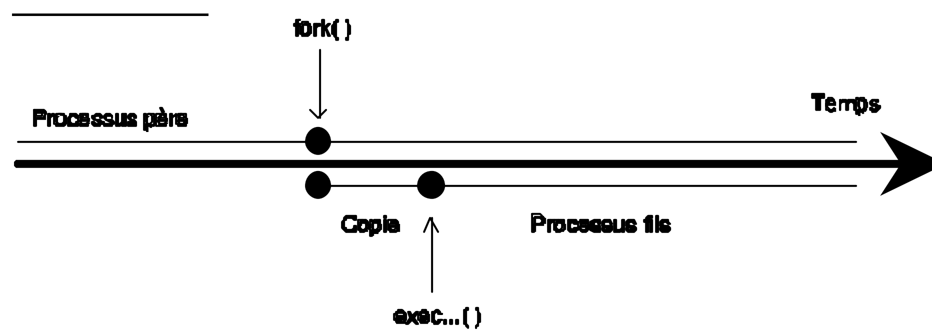
- Le père est la stricte continuation du processus appelant.
- Le fils est une copie du père, mais en tant que nouveau processus il dispose de son propre pid, seule chose qui le distingue de son père.

Il peut paraître absurde en première lecture de créer un second processus copie du premier !

Pour que cela devienne vraiment intéressant il faut que le processus fils réalise aussitôt un appel à exec...( ) pour substituer à cette copie un autre processus.

On parvient ainsi au résultat escompté.

schéma de fork( ) + exec...( ):



Le processus père poursuit son exécution après le fork( ), tandis que le fils commence son exécution après ce même fork( ).

Père et fils sont à même de se reconnaître par le code de retour que leur transmet la fonction fork( ):

- Le père reçoit du fork( ) le pid > 0 attribué par le système à son fils.
- Le fils reçoit du fork( ) la valeur 0.

En cas d'échec de la fonction fork( ), celle-ci retourne -1 au père et il n'y a pas de copie fils créée.

Exemple fork( ) + exec...( ):

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void main( )
	{
	int pid;
	char strfils[ ]= "Je suis le fils\n";
	char strpere[ ]= "Je suis le père\n";

	printf( strpere );
	pid= fork( );
	if ( pid == -1 )
		{
		printf( "Erreur lors du fork.\n" );
		exit( 1 );
		}
```

```
        if ( pid == 0 )   /* code exécuté par le fils */
                {
                printf( strfils );
                execl( "sonny", "sonny", NULL);
                /* le fils continue et fait sa vie */
                }
        else              /* code exécuté par le père */
                {
                printf( strpere );
                ...
                /* le père continue sa vie */
                }
}
```

3) Terminaison d'un processus:

Un processus se termine normalement par un appel explicite à exit( ).
Toutefois la fin de la fonction main( ) marque aussi la fin du programme et provoque un appel implicite à exit( ).

Syntaxe: exit(int code);

On passe à exit( ) un code de retour destiné au programme père—le shell en général.
Conventionnellement code=0 signifie que tout s'est bien passé et un code non nul signale un problème.
En cas d'exit( ) implicite le code de retour est 0.

Le processus père peut comme on le verra ci-après prendre connaissance du code de retour de chacun de ses fils terminés.

4) L'appel à wait...( ):

La combinaison fork( ) + exec...( ) ne répond pas encore à tous les besoins:
souvent le processus père souhaite suspendre son activité et attendre la fin du processus fils qu'il a engendré.
C'est le cas du shell lorsqu'il exécute une commande en avant plan, tandis que lors d'une tâche
de fond le shell poursuit son exécution et passe à la commande suivante.

Plusieurs fonctions permettent de suspendre le processus père jusqu'à la terminaison d'un ou plusieurs de ses fils; il s'agit de wait( ), waitpid( ) et waitid( ).

Consacrons-nous à waitpid( ) par exemple:

Syntaxe: waitpid(int pid_fils, int *val_ptr, 0);

Le premier paramètre est le pid du fils dont le père souhaite attendre la fin, tel qu'il est retourné par la fonction fork( ).

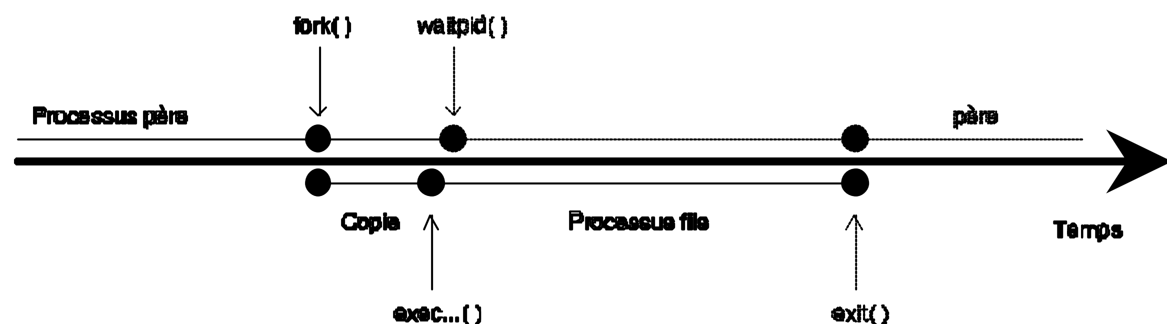
Le second paramètre val_ptr est l'adresse d'une variable int dont on pourra extraire le code de retour du processus fils à l'aide de macro-fonctions prédéfinies:

Syntaxe: int WIFEXITED(int val);
int WEXITSTATUS(int val);

WIFEXITED(val) retourne un résultat non nul si le fils s'est terminé normalement.
WEXITSTATUS(val) extrait de val le code de retour de ce processus fils.

Le dernier paramètre de waitpid( ) qui ne présente pas d'intérêt pour nous est ici fixé à zéro.

Schéma de fork( ) + exec...( ) + waitpid( ):



Un processus fils passe à l'état de zombie lorsqu'il se termine par un appel à exit( ) avant même que le processus père n'ait fait appel à wait...( ) !

Remarques: Lors de l'exécution d'un script shell, le shell principal effectue un fork( )

et confie à sa copie ainsi crée la tâche d'interpréter ledit script; le shell principal se contente d'attendre la fin du shell secondaire.

S'il s'agit d'un C-shell chargé d'exécuter un script Bourne shell, alors le fils effectue de plus un exec...( ) de sh avec en paramètre le nom du script à exécuter.

Exemple fork( ) + exec...( ) + wait...( ):

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void main( )
        {
        int pid, code;
        char strfils[ ]= "Je suis le fils\n";
        char strpere[ ]= "Je suis le père\n";
```

```
        printf( strpere );
        pid= fork( );

        if ( pid == -1 )
                {
                printf( "Erreur lors du fork.\n" );
                exit( 1 );
                }

        if ( pid == 0 )   /* code exécuté par le fils */
                {
                printf( strfils );
                execl( "sonny", "sonny", NULL);
                /* le fils continue et fait sa vie */
                }
        else              /* code exécuté par le père */
                {
                waitpid(pid, &code, 0);
                code= WEXITSTATUS(code);
                printf( strpere );
                printf("Fils terminé avec code %d", code);
                ...
                /* le père continue sa vie */
        }
        }
```

5) La récupération des paramètres:

On a vu comment passer des paramètres à un programme, mais reste à savoir comment le programme récupère ces paramètres en C
(qu'ils viennent d'un exec...( ) ou directement de la ligne de commande).

On déclare en général dans le programme la fonction main( ) sans paramètre.

Si on veut récupérer d'éventuels paramètres on doit déclarer main(int argc, char **argv) où argc sera le nombre de paramètres transmis et argv un tableau de pointeurs sur ces paramètres sous forme de chaînes de caractères accessibles par :
argv[∅] , ..., argv[argc-1].

Remarque: char **argv est souvent remplacé par char *argv[ ], ce qui revient au même, désignant un tableau de pointeurs sur caractères.

Exemple:

```
int main( int argc, char **argv )
        {
        char nom[128];
        ...
        strcpy( nom, argv[∅] );
        printf( ″je suis le programme %s ″, nom );
        /* argv[∅] est conventionnellement le nom du programme */
        ...
        exit( ∅ );        /* fin normale du programme */
        }
```

6) Fonctions de récupération de pid:

Les fonctions getpid( ) et getppid( ) permettent respectivement à un processus d'accéder à son propre pid et au pid de son père:

Syntaxe:

```
int getpid( );
int getppid( );
```

7) Manipulation de l'environnement:

Les fonctions getenv( ) et putenv( ) permettent à un processus d'accéder à ses variables d'environnement, héritées notamment de son père, ou transmises par celui-ci:

Syntaxe:
```
char * getenv( char *name );
int putenv( char *string );
```

D'une part getenv( ) permet de récupérer le contenu de la variable name.
Elle renvoie un pointeur sur la chaîne contenue dans cette variable ou un pointeur NULL si ladite variable n'existe pas.

D'autre part putenv( ) sert à modifier ou ajouter une variable d'environnement sous la forme "name=chaine"

Si la variable existe déjà son contenu est modifié.
putenv( ) retourne 0 si tout va bien et une valeur non nulle en cas d'impossibilité.

8) Appel system( ):

La fonction system( ) remplace l'appel à fork( ) + exec...( ) + waitpid( ) et en étend les possibilités à certains égards.

Syntaxe:
```
int system( char *commande );
```

La commande passée en paramètre est exécutée par un shell telle qu'elle serait exécutée en ligne de commande.

Exemple:
```
system( "ls -a -l /usr/bin > toto" );
```

Remarques: En fait un tel appel est plus lourd qu'un appel direct à fork( ) + exec...( ) + wait...( ) dans la mesure où un shell intermédiaire est généré pour exécuter la commande. Toutefois il y a des avantages dans la mesure où l'on peut très facilement rediriger la sortie standard du processus dans un fichier sans avoir à programmer soi-même la chose...

9) Redirection des entrées / sorties:

La redirection de l'entrée ou de la sortie standard passe par la fonction dup2( ) dont le rôle consiste à remplacer dans la table des descripteurs un descripteur utilisé par un autre descripteur.

Ainsi toutes les opérations faites ultérieurement sur le fichier qui était associé au descripteur d'origine auront en réalité lieu sur le fichier associé au second descripteur !

Afin de restaurer le contexte initial après le détournement, on prendra soin de mémoriser au préalable le descripteur modifié, par duplication avec la fonction dup( ).

Syntaxe:
```
int dup( int desc );
int dup2( int desc2, int desc1 );
```

dup( ) retourne le nouveau descripteur dupliquant l'ancien ou -1 en cas d'erreur.
dup2( ) retourne -1 en cas d'erreur (descripteurs non valides ...).

Remarque: Pour rediriger l'entrée standard, la sortie standard ou la sortie erreur standard
il suffit de donner à desc1 la valeur 0, 1 ou 2 correspondant à ces entrées particulières de la table des descripteurs.

Exemple:

```
#include <stdio.h>
#include <unistd.h>

#define STD_OUT  1

int fd, output;

void main( )
        {
        printf( "Coucou !\n" );  /* affiche à l'écran */

        fd= creat( "toto", 0777 );          /* crée un fichier */
        output= dup( STD_OUT );      /* mémorise stdout */
        dup2( fd, STD_OUT );            /* redirige la sortie dans le fichier */
        printf( "Vous ne verrez pas ce message !\n" );
                                                    /* envoyé en fait dans le fichier */
        dup2( output, STD_OUT );     /* restaure stdout */
        close( fd );                              /* ferme le  fichier */

        printf( "Re coucou !\n" );          /* à nouveau à l'écran */
        }
```

IV) Les signaux.

1) Le principe des signaux:

Un signal est une information envoyée à un processus pour attirer son attention
sur un événement particulier.
Un tel signal peut provenir d'un autre processus, du noyau du système UNIX, ou encore
faire suite à une action de l'utilisateur au niveau du terminal.

Il existe 32 signaux correspondant chacun à un type d'événement très précis.
Pour chaque processus en cours d'exécution le système gère une table de 32 bits
et une table de 32 adresses associées.

Lorsque tel signal est envoyé à un processus, le bit correspondant est levé dans sa table
des signaux. Le processus n'en aura connaissance que lorsqu'il sera **élu**; un signal est donc
reçu de façon asynchrone. Un signal en attente est dit pendant.

Lorsque le signal est détecté, le processus est détourné temporairement pour exécuter
la tâche associée à ce signal, référencée par l'entrée correspondante de la table
des adresses.

Pour chaque signal il y a plusieurs types d'actions possibles:

- Ignorer le signal (Ignore)
- Effectuer un traitement par défaut qui peut être
  - terminer le processus (Exit)
  - terminer le processus et générer un fichier core (Core)
  - suspendre / reprendre l'exécution du processus
- Effectuer un traitement spécifique défini par le programmeur

2) Les signaux du noyau:

Il arrive que le kernel soit obligé d'interrompre un processus devenu chaotique
ou ayant engendré une erreur grave (Erreur d'adressage mémoire, division par zéro...).
Pour cela le kernel dispose de plusieurs signaux très précis.

Le traitement par défaut associé à ces signaux consiste en général à terminer le processus
en générant une image core: c'est à dire un fichier nommé core qui est une image mémoire
du processus incriminé et qui peut être utilisé lors du debugage.

Chaque signal a un nom, associé à une constante prédéfinie dans <signal.h>.
Ces constantes varient de 1 à la valeur maximale NSIG-1:

SIGNAL	Signification
SIGILL	Instruction processeur non valide
SIGFPE	Division par zéro
SIGSEGV	Violation d'espace mémoire
SIGXCPU	Epuisement du temps CPU
SIGFSZ	Dépassement de la taille maximale pour un fichier
SIGSYS	Erreur d'appel système
SIGALRM	Signal d'alarme programmé
SIGPIPE	Ecriture dans un pipe sans lecteur

3) Les signaux du terminal:

Un processus lancé depuis un terminal—son terminal de contrôle—peut recevoir certains signaux lorsque l'utilisateur effectue des actions particulières sur ce terminal.

Ainsi l'appui sur la touche Break (intr=^C) a pour effet de générer un signal de terminaison vers le processus en cours. C'est ce qui permet d'interrompre brutalement une commande.

De même la combinaison ^Z (susp=^Z) envoie au processus en cours le signal de suspension.
Le processus pourra être repris ultérieurement avec un signal de reprise.

Les signaux du terminal sont:

SIGNAL	Signification
SIGINT	Signal de terminaison intr
SIGQUIT	Signal de terminaison avec génération d'un core
SIGTSTP	Signal de suspension susp
SIGCONT	Signal de reprise après suspension
SIGUP	Signal de déconnexion

4) Les signaux logiciels:

Les signaux logiciels sont ceux envoyés par un processus à un autre processus, dans un but de dialogue (certes très limité !).

SIGNAL	Signification
SIGTERM	Signal de terminaison immédiate Généré par la commande kill -15 pid

SIGKILL	Signal de terminaison immédiate et inconditionnelle Généré par la commande kill -9 pid Ne peut être ni ignoré ni détourné.
SIGCHLD	Signal de transition d'état émis par un processus fils
SIGUSR1 SIGUSR2	Signaux disponibles pour le programmeur

5) Primitives de gestion des signaux:

La gestion des signaux requiert l'inclusion du header <signal.h>.

Les signaux logiciels sont générés par la primitive kill( ):

Syntaxe: int kill(int pid, int SIG);

Le pid est celui du processus à qui est destiné le signal et SIG est l'une des constantes vues plus haut et définies dans <signal.h>

La réception du signal n'est pas immédiate et il faut en tenir compte dans la programmation ! Le même signal émis plusieurs fois de suite ne sera pas forcement reçu plusieurs fois par le processus visé en raison du délai de latence avant prise en compte du signal.

La primitive kill( ) retourne -1 en cas d'erreur (*pid* non valide...).

Chaque signal est associé à une disposition par défaut. Il est possible—sauf exceptions— de modifier le traitement associé à un signal en définissant un traitement spécifique. On utilise à cette fin la primitive signal( ):

Syntaxe: signal(int SIG, void (* disp(int)) ( ));

Dans cette déclaration alambiquée, *SIG* est bien sûr la constante définissant le signal dont on veut modifier le traitement; quant au reste, il s'agit de l'adresse d'une fonction recevant un paramètre de type int et void en retour !

En pratique il suffit de passer le nom de la fonction devant assurer le traitement, qui est effectivement une adresse quelque part dans le code...
On peut aussi utiliser un traitement prédéfini désigné par une macro-fonction:

SIG_IGN	Pour ignorer le signal
SIG_DFL	Pour restaurer le traitement par défaut

Lorsque l'on opte pour un traitement spécifique par la fonction disp( ), cette fonction disp( ) doit être définie dans le programme et déclarée sous la forme:

```
static void disp( int );
```

En effet cette fonction recevra lors du traitement futur d'un signal un paramètre entier de la part du noyau, représentant le numéro du signal concerné.

(Cela permet à une seule fonction de gérer éventuellement plusieurs signaux).

💣 Attention: Certains signaux ne peuvent être ni ignorés ni détournés,
tels que SIGSEGV et SIGKILL.

Excepté pour SIGILL et quelques autres, avant l'exécution du traitement associé par la fonction disp( ), le signal est repositionné à SIG_DFL, afin d'éviter des appels imbriqués à la fonction disp( ).

Il est donc souhaitable, à l'intérieur de la fonction disp( ):

- d'ignorer tout nouvel appel en positionnant le signal sur SIG_IGN
  dès le début du traitement.

- de repositionner le signal sur disp( ) à la fin du traitement
  pour le prochain appel.

Remarque: Lors d'un fork( ) les dispositions associées aux signaux sont conservées.
Par contre les signaux détournés sont restaurés à SIG_DFL lors d'un exec...( )
puisque les fonctions de traitement spécifiques sont démantelées avec le processus !

La fonction disp( ) associée à un signal particulier peut se contenter de positionner une variable globale du programme, qui sera analysée à un moment plus opportun dans le déroulement
normal de ce programme.

Signalons la fonction psignal( ) qui affiche un message associé au numéro de signal qui lui est communiqué, complété par un commentaire personnel si nécessaire.
Elle est disponible à partir de System V Release 4:

Syntaxe: psignal(int SIG, char *comment);

Si le numéro de signal n'est pas entre 1 et N_SIG-1 un message d'erreur est affiché

Le temps et les signaux:

Plusieurs fonctions traitent du temps et des signaux. Signalons alarm( ), pause( ) et sleep( ):

Syntaxe:
```
unsigned alarm( unsigned seconds );
pause ( void );
unsigned sleep( unsigned seconds );
```

La fonction alarm( ) demande au kernel de générer un signal SIGALRM à l'attention du processus appelant au bout d'un certain nombre de secondes.
Un nouvel appel annule le précédent, et dans ce cas la fonction retourne le temps résiduel de l'appel précédent.

Si *seconds*=0 l'alarme est annulée.
L'utilisation d'alarm( ) suppose de détourner le signal SIGALRM vers un traitement spécifique.

La fonction pause( ) suspend l'exécution du processus jusqu'à l'arrivée d'un signal.
Le processus se poursuivra dès la réception d'un signal non ignoré.

Enfin sleep( ) suspend le processus pour un certain nombre de secondes, ou jusqu'à la réception d'un signal non ignoré.
La durée du sleep( ) est rendue incertaine par l'arrivée possible d'un signal...
Cette fonction retourne la durée effective de sommeil, ce qui permet grâce à une boucle de réaliser tout de même la temporisation voulue.

Exemple général:

Ce programme détourne tous les signaux qui peuvent l'être vers une même fonction, qui affiche un message pour chaque signal reçu.
On pourra faire tourner ce processus en tâche de fond et lui envoyer toutes sortes de signaux avec la commande kill...

```
#include <stdio.h>
#include <string.h>
#include <signal.h>

#define STDOUT 1

void disp( int sig );

char msg[80];

static void disp( int sig )
        {
        signal( sig, SIG_IGN );          /* inhibe appels imbriqués */
        sprintf( msg, "Signal recu: %d\n", sig );
        write( STDOUT, msg, strlen( msg ) );
        signal( sig, disp );             /* repositionne le signal */
        }

main( )
        {
        int s;

        printf( "Détournement de tous les signaux\n" );
        for (s=1; s < NSIG ; s++ )
                signal( s, disp );

        printf( "Tuez moi pour en finir !\n" );
        while(1);                        /* boucle infinie */
        }
```

V) Curses et terminfo.

1) Le principe:

On a vu que les terminaux texte fonctionnent en mode plein écran à l'aide
de séquences de commande (séquences Escape ou séquences de Contrôle).

Les commandes utilisées par des milliers de terminaux sont stockées dans la base
de données terminfo.

De nombreux systèmes de développement en C sous UNIX proposent
une bibliothèque de fonctions dédiée à la gestion du mode plein écran, qui permet
de programmer des applications portables fonctionnant sur tous les terminaux standards.

Cette bibliothèque se nomme **curses**.

On doit pour l'utiliser:

- Inclure le header <curses.h> dans le source C.
- Linker la librairie curses au programme.

On utilisera en général un makefile approprié (cf § Outils de développement).

Les fonctions de cette bibliothèque gèrent à la fois la saisie et l'affichage,
qui ne peuvent plus se faire avec printf( ) et scanf( ) dès lors qu'on passe en mode
curses.

La bibliothèque curses s'appuie sur la variable d'environnement TERM ou term,
qui doit être correctement renseignée, pour déterminer lors de l'exécution le type
de terminal à piloter.
Ce type doit être référencé dans la base de données terminfo.
Chaque action possible à l'écran correspond à un mnémonique auquel terminfo
associe une séquence de commande pour le terminal indiqué.
De même les touches spéciales du clavier sont désignées par des mnémoniques
auxquels on associe les séquences de commande appropriées.

Exemples: La commande d'effacement d'écran a pour mnémonique clear,
clear=^L sur un DEC vt220.
La commande de positionnement absolu du curseur est cup,
cup=ESC[α,βH sur un DEC vt220 avec (α,β) les coordonnées
du curseur à partir du coin supérieur gauche.
La touche de fonction F1 est désignée par kf1,
kf1=ESCOP sur DEC vt220.

Tous les terminaux ne supportent pas toutes les fonctionnalités existantes—par exemple
les terminaux monochromes ne supportent pas la couleur. Seules les fonctionnalités

supportées par le terminal utilisé sont effectivement associées à une séquence de commande.
Les autres sont en général ignorées.

2) Gestion de l'écran en mode curses:

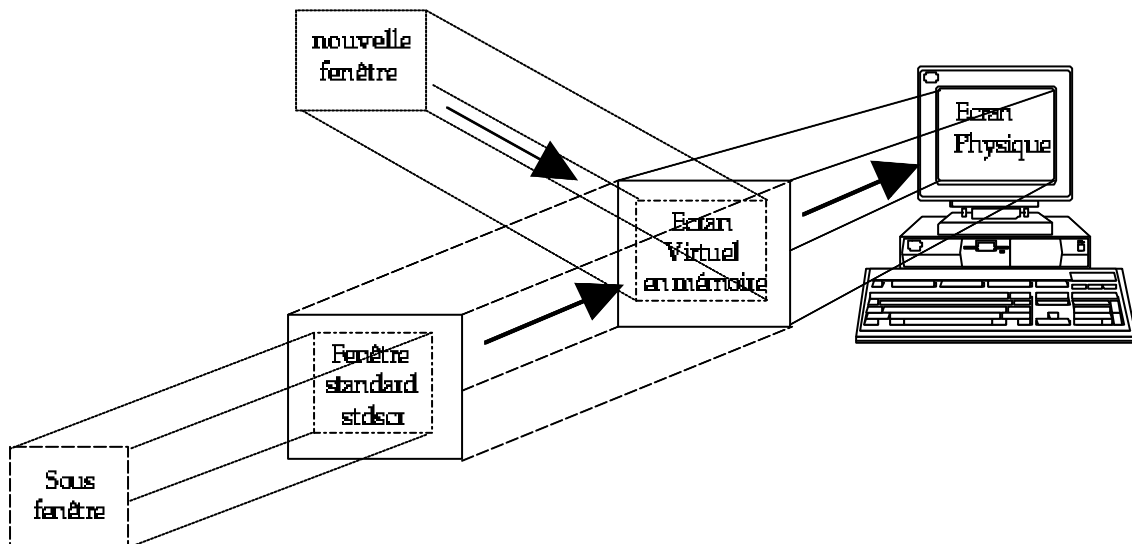
Les fonctions de la bibliothèque curses ne travaillent pas directement à l'écran:
elles utilisent en fait un **écran virtuel** qui est une zone mémoire reproduisant l'écran
et où l'on mémorise pour chaque caractère à l'écran le caractère lui-même mais aussi
son attribut.
Cela est rendu nécessaire par le fait que le système UNIX ne peut accéder à la mémoire
vidéo du terminal pour savoir ce qui est affiché, aussi doit-on mémoriser dans l'écran
virtuel une représentation de l'affichage.

Les fonctions de la bibliothèque curses n'agissent que sur l'écran virtuel, dont les
modifications ne sont transmises au terminal que lors d'une opération explicite de
rafraîchissement de l'écran, que l'on peut faire à tout moment.
En général on réalise dans l'écran virtuel plusieurs modifications, puis on rafraîchit l'écran,
car l'opération de rafraîchissement, bien qu'optimisée est parfois laborieuse en raison
de la faible vitesse de communication entre le serveur et le terminal.

La bibliothèque curses gère de plus des **fenêtres** à l'intérieur de l'écran virtuel.
La fenêtre active par défaut est la **fenêtre standard** qui correspond à l'écran virtuel tout
entier.
Une opération réalisée dans une fenêtre n'affecte pas non plus immédiatement l'écran
virtuel.
Il faut à nouveau explicitement **rafraîchir l'écran virtuel** pour que la mise à jour ait lieu.

Enfin, dans une fenêtre on peut définir des **sous fenêtres**:
les modifications réalisées dans une sous fenêtre affectent directement la fenêtre mère.

Schéma de principe:



3) Initialisation du mode curses:

Un programme souhaitant utiliser les fonctions de la bibliothèque curses doit initialiser le mode plein écran par un appel à initscr( ).
Dans cette phase d'initialisation la fonction recherche la variable TERM ou term et teste si le terminal indiqué est connu de terminfo.
A défaut le mode curses n'est pas initialisé et la fonction affiche un message d'erreur.

Si tout va bien le mode plein écran est initialisé et on peut faire appel aux fonctions que nous allons détailler.

A la fin du programme on doit faire appel à endwin( ) pour revenir en mode ligne, sans quoi le terminal risque de se bloquer...

Syntaxe:

```
initscr( );
endwin( );
```

Entre l'appel à initscr( ) et l'appel à endwin( ) on ne peut plus utiliser le terminal en mode ligne
et on ne doit plus faire appel à printf( ) et scanf( ).

4) Manipulations dans la fenêtre standard:

a) Saisie et affichage:

Voyons pour commencer ce que l'on peut faire dans la fenêtre standard, qui correspond rappelons-le à l'écran virtuel tout entier, donc à l'écran physique tout entier.
L'écran présente en général 80 colonnes et 24 ou 25 lignes.
On peut accéder aux dimensions de l'écran par les variables globales LINES et COLS

qui sont initialisées par initscr( ).

La fenêtre standard est nommée stdscr. On peut y afficher toutes sortes de données avec les fonctions addch( ), addstr( ) et printw( ), et y saisir des données avec les fonctions getch( ), getstr( ) et scanw( ).
Ceci en tout point de la fenêtre grâce au curseur d'affichage que l'on déplace avec move( ).

Un type de variable particulier a été introduit, le type chtype, qui permet de gérer à la fois un caractère et son attribut à l'écran.

Syntaxe:

```
move( int y, int x );

addch( chtype chr );
addstr( char *string );
printw( char *format, var1, ... , varN);
```

Les coordonnées à l'écran sont passées à move( ) dans l'ordre ligne puis colonne; ces coordonnées commencent à (0, 0) pour le coin supérieur gauche.

On peut bien sûr passer à addch( ) un simple char pour affichage, à la position actuelle du curseur.
De même addstr( ) affiche une chaîne de caractères à la position du curseur, et printw( ) est l'équivalent curses de printf( ), avec la même syntaxe.

La position du curseur est modifiée après chaque affichage.

💣 Attention: Ces fonctions n'ont aucun effet à l'écran tant que celui-ci n'est pas rafraîchi explicitement ! (Voir ci-après).

Syntaxe:

```
chtype getch( );
getstr( char *string );
scanw( char *format, &var );
```

La fonction getch( ) permet la saisie de caractères au clavier mais aussi sous certaines conditions la détection des touches spéciales du clavier (Fonctions, flèches ...).
Pour pouvoir détecter ces touches spéciales il est nécessaire de faire appel au préalable à keypad( ) sous la forme keypad(stdscr, TRUE);

On utilise alors des constantes prédéfinies pour reconnaître ces touches:

KEY_DOWN	flèche bas	KEY_HOME	home
KEY_UP	flèche haut	KEY_BACKSPACE	backspace
KEY_LEFT	flèche gauche	KEY_F(1)	touche F1
KEY_RIGHT	flèche droite	...	

Consulter man curses pour avoir toute la liste des constantes associées.

D’autre part la fonction getstr( ) réalise la saisie d’une chaîne de caractères
et scanw( ) est l’équivalent curses de scanf( ), avec la même syntaxe.

Contrairement aux fonctions d’affichage dont l’effet n’est pas immédiat à l’écran,
ces fonctions de saisie agissent immédiatement à l’écran car il est en général indispensable
que l’utilisateur voie ce qu’il tape au clavier !
En fait on peut modifier cette disposition à l’aide des fonctions echo( ) et noecho( ).

Le curseur d’affichage est déplacé au fil de la saisie.

Afin de tirer profit de ces fonctions, ajoutons la fonction refresh( ), dont le rôle consiste
à mettre à jour l’écran virtuel à partir de la fenêtre standard, puis à rafraîchir l’écran physique
à partir de l’écran virtuel, c’est à dire à rendre visibles les modifications réalisées !

Syntaxe: refresh( );

L’effacement de la fenêtre standard est réalisé pat clear( ):

Syntaxe: clear( );

L’effacement n’est encore une fois effectif qu’après un appel à refresh( )

b) Gestion des couleurs et attributs:

Chaque caractère à l’écran dispose d’attributs qui peuvent être:
(selon les possibilités du terminal)

- clignotement
- graissage
- soulignement
- inversion vidéo
- couleur du caractère
- couleur du fond

Ces attributs sont mémorisés dans la fenêtre standard et dans l’écran virtuel pour chaque caractère de l’écran.

Les fonctions attrset( ), attron( ) et attroff( ) permettent de positionner les attributs
à utiliser pour les prochaines opérations de saisie / affichage:

Syntaxe:
```
attrset( chtype attrib );
attron( chtype attrib );
attroff( chtype attrib );
```

Plus précisément, attrset( ) positionne exactement les attributs demandés dans attrib,
tandis que attron( ) ajoute les attributs demandés aux attributs existant et attroff( ) les enlève.

Le paramètre attrib est une disjonction de constantes prédéfinies telles que:

A_STANDOUT	graissage
A_UNDERLINE	soulignement
A_BLINK	clignotement
A_REVERSE	inversion vidéo

Si un attribut n'est pas supporté par le terminal il sera en général ignoré.

Exemple: attrset(A_BLINK ¦ A_STANDOUT);

Pour ce qui est des couleurs, il convient d'abord de déterminer si le terminal supporte ou non
la couleur avec has_colors( ), qui renvoie TRUE ou FALSE selon les possibilités du terminal.
S'il s'agit bien d'un terminal couleur, on doit initialiser le mode couleur avec start_color( ):

Syntaxe:

```
bool has_colors( );
start_color( );
```

La fonction start_color( ) initialise les variables globales COLORS et COLOR_PAIRS
qui contiennent le nombre de couleurs supportées et le nombre de paires de couleurs
supportées pour un caractère et le fond sur lequel il est écrit.

En général COLORS=8 et COLOR_PAIRS=COLORS²=64.

Les 8 couleurs de base sont définies par des constantes:

COLOR_BLACK	0	Noir
COLOR_BLUE	1	Bleu
COLOR_GREEN	2	Vert
COLOR_CYAN	3	Cyan
COLOR_RED	4	Rouge
COLOR_MAGENTA	5	Magenta
COLOR_YELLOW	6	Jaune
COLOR_WHITE	7	Blanc

Or un attribut de couleur complet doit comprendre une couleur pour le caractère
et une couleur pour le fond sur lequel il sera écrit, c'est à dire une paire de couleurs.
Une telle paire de couleur sera désignée par un numéro de 0 à COLOR_PAIRS-1,
selon une distribution à définir à l'aide de init_pair( ):

Syntaxe: init_pair(short numpair, short colfg, short colbg);

Le plus simple est de faire une double boucle où l'on fait varier les couleurs du texte et du fond de 0 à COLORS-1 pour définir le tableau interne des paires de couleurs:

Exemple:

```
int i, j;

for ( i= 0 ; i< COLORS ; i++ )
        for ( j= 0 ; j< COLORS ; j++ )
                init_pair( 8*i+j, i, j );
```

On peut alors positionner une paire de couleur pour les prochains affichages à l'aide de la fonction attrset( ) et de la macro-fonction COLOR_PAIR( ) qui fabrique un attribut à partir d'une paire de couleurs:

Syntaxe: chtype COLOR_PAIR(short numpair);

Exemple:

```
attrset( COLOR_PAIR( 061 ) );
attrset( COLOR_PAIR( 8*COLOR_YELLOW+COLOR_BLUE) );
```

Ces deux formes sont équivalentes vu la définition du tableau des paires de couleurs. La forme octale 061 est plus courte et néanmoins assez parlante puisqu'on y voit que la couleur de texte est 6=Jaune et la couleur de fond 1=Blue.

5) Fenêtres et sous fenêtres:

On peut définir une nouvelle fenêtre, représentant seulement une partie de l'écran virtuel, pour des opérations particulières, avec newwin( ):

Syntaxe: WINDOW * newwin(int nblig, int nbcol, int y0, int x0);

On fournit à newwin( ) les coordonnées (y0, x0) du coin supérieur gauche de la fenêtre ainsi que le nombre de lignes et de colonnes de la fenêtre.

newwin( ) retourne alors un pointeur sur une structure de type prédéfini WINDOW contenant les paramètres nécessaires à la gestion de la fenêtre et une zone de travail représentant la partie concernée de l'écran virtuel.

Les manipulations effectuées ultérieurement sur cette fenêtre passeront par le pointeur ainsi obtenu, que l'on mémorisera soigneusement jusqu'à la destruction de la fenêtre.

La zone de travail est initialisée à la création à partir de la région correspondante de l'écran virtuel.

Les opérations réalisées dans cette fenêtre n'affectent que la zone de travail propre à la fenêtre
et ne seront reportées dans l'écran virtuel et sur l'écran physique qu'après rafraîchissement.

Les zones de travail des autres fenêtres, même si elles correspondent à des régions de l'écran
qui chevauchent la zone utilisée ne seront pas modifiées.
On peut ainsi mémoriser dans une fenêtre toute une partie de l'écran, puis la surcharger temporairement en travaillant dans la fenêtre standard, et enfin restaurer le contexte initial à partir de la fenêtre mémorisée.

Par ailleurs, on peut définir une sous-fenêtre à l'intérieur d'une fenêtre avec subwin( ):

Syntaxe: WINDOW * subwin(WINDOW *w_ptr, int nblig, int nbcol, int y0, int x0);

En plus des dimensions et coordonnées de la sous-fenêtre exprimées de façon absolue par rapport au coin de l'écran, on passe à subwin( ) le pointeur sur la fenêtre mère à laquelle la sous fenêtre sera rattachée:

Exemple:

```
WINDOW *w_ptr;

w_ptr= subwin( stdscr, 10, 60, 10, 10 );
```

En effet stdscr est lui-même un pointeur de type WINDOW * associé à la fenêtre standard !

La différence majeure entre une nouvelle fenêtre et une sous fenêtre est qu'une sous fenêtre partage la zone de travail de sa fenêtre mère.
Par conséquent toutes les modifications faites dans une sous fenêtre affectent immédiatement la région correspondante de la fenêtre mère.

On utilise en général une sous-fenêtre pour réaliser des opérations dans une partie d'une fenêtre
en étant sûr de ne pas affecter le reste de cette fenêtre.

Le rafraîchissement de l'écran virtuel à partir d'une fenêtre ou d'une sous-fenêtre passe par la fonction wnoutrefresh( ).

Le rafraîchissement de l'écran physique à partir de l'écran virtuel passe par doupdate( ).
On peut réaliser les deux en appelant directement wrefresh( ):

Syntaxe: wnoutrefresh(WINDOW *w_ptr);

doupdate( );
wrefresh(WINDOW *w_ptr);

remarques: refresh( ) équivaut en fait à wrefresh(stdscr);

En fin de programme ou dès qu'elles ne sont plus utiles les fenêtres et sous-fenêtres crées doivent être détruites par delwin( ):

Syntaxe: delwin(WINDOW *w_ptr);

Après l'appel à delwin( ) on ne doit plus utiliser w_ptr.

Les fonctions d'affichage et de saisie dans une fenêtre ou sous-fenêtre sont semblables en tous points à celles vues pour stdscr; elles commencent par un w et acceptent un paramètre
supplémentaire désignant la fenêtre concernée par l'opération:

Syntaxe:

```
waddch( WINDOW *w_ptr, chtype chr );
waddstr( WINDOW *w_ptr, char *string );
wprintw( WINDOW *w_ptr, char *format, var1, ... , varN );

chtype wgetch( WINDOW *w_ptr );
keypad( WINDOW *w_ptr );
wgetstr( WINDOW *w_ptr, char *string );
wscanw( WINDOW *w_ptr, char *format, &var );

wmove( WINDOW *w_ptr, int y, int x );
```

Le maniement des couleurs et attributs est identique et s'appuie sur les fonctions suivantes:

Syntaxe:

```
wattrset( WINDOW *w_ptr, chtype attrib );
wattron( WINDOW *w_ptr, chtype attrib );
wattroff( WINDOW *w_ptr, chtype attrib );
```

On efface une fenêtre ou sous-fenêtre avec wclear( ):

Syntaxe: wclear(WINDOW *w_ptr);

Remarque: Il existe bien d'autre fonctions dans la bibliothèque curses.
Consulter la documentation et le man en cas de besoin.

Il existe notamment des fonctions de bas niveau capables d'extraire des renseignement directement de la base de données terminfo; elles sont implicitement utilisées par les fonctions
vues ci-dessus.

VI) Les pipes.

1) Description:

On a vu que des processus peuvent communiquer de façon rudimentaire en échangeant des signaux. Des mécanismes plus sophistiqués existent, regroupés sous le terme d'IPC (Internal Process Communication):

- Les sémaphores.
- Les files de messages.
- Les segments de mémoire partagée.

- Les tubes ou pipes.

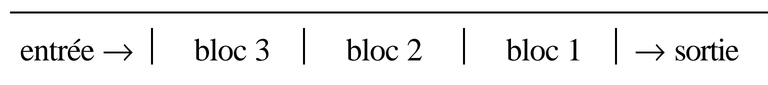
Les trois premier moyens de communication permettent des échanges d'informations en mémoire. Le dernier mécanisme, celui des pipes, est un moyen de communication externe reposant sur des fichiers d'un type particulier: les **tubes** ou **pipes**.

Nous allons développer ici l'utilisation de ces pipes.

Un pipe est un fichier spécial de type fifo (first-in first-out) muni en fait de deux extrémités:

- une entrée
- une sortie

d'où le schéma suivant illustrant un pipe:



fichier fifo

En général deux processus qui veulent échanger des informations par un pipe ouvrent ce pipe
chacun à une extrémité, l'un en écriture pour envoyer des données dans le pipe, l'autre en lecture pour les récupérer à l'autre extrémité.

Les opérations d'écriture et de lecture se feront comme dans un fichier, sauf que la lecture est destructrice dans la mesure où les données lues sont extraites du pipe.

Un pipe procure donc une communication unidirectionnelle. Il en faut deux pour pouvoir échanger des données dans les deux sens.

Il existe en fait deux types de pipes:

- Les pipes locaux.
- Les pipes nommés.

Un pipe local est limité à l'échange entre des processus père et fils—ou ayant du moins un ascendant commun, mais c'est déjà plus compliqué—et n'est pas matérialisé par une entrée dans le système de fichiers.

Par contre un pipe nommé permet d'échanger des données entre deux processus quelconques
et est matérialisé par un fichier spécial de type fifo dans le système de fichiers.

La taille d'un pipe est limitée à une valeur PIPSIZ, constante définie dans <limits.h>,
aussi ne faut-il pas remplir inconsidérément un pipe...

2) Les pipes locaux:

Un tube local est donc principalement utilisé pour faire communiquer des processus père et fils.
On crée un tel pipe local à partie de la fonction pipe( ):

Syntaxe: int pipe(int desc[2]);

On lui passe l'adresse d'un tableau de deux entiers, qui sera complété par la fonction au retour en cas de succès; en cas d'échec la fonction pipe( ) retourne -1.
Lorsque l'opération réussit, le tableau contient deux descripteurs:

desc[0]	handle pour l'accès en lecture sur le pipe (sortie) .
desc[1]	handle pour l'accès en écriture sur le pipe (entrée).

On voit alors qu'il est nécessaire qu'existe un lien de parenté entre les processus utilisateurs dans la mesure où le processus qui crée le pipe transmettra les descripteurs obtenus
par héritage de la table des descripteurs des fichiers ouverts (cf § Gestion des processus).

Il est donc impératif que le processus père fasse appel à pipe( ) avant l'appel à fork( ) !

Ensuite les opérations de lecture et écriture se font comme dans un fichier avec les fonctions read( ) et write( ) auxquelles on transmet le descripteur adapté desc[0] ou desc[1]:

Syntaxe: int read(int desc, char *tampon, int nb);
int write(int desc, char *tampon, int nb);

Le processus qui envoie des informations écrit en général un bloc d'octets dans le tube,
qui vient s'ajouter aux données existantes.
Le processus lecteur lit également un bloc de données mais il ne lui est pas possible

de savoir comment les octets étaient initialement groupés lors de l'écriture.
Pour pallier ce problème les deux processus peuvent convenir d'un caractère particulier jouant le rôle de délimiteur si nécessaire.

Attention: La lecture peut être bloquante ou non bloquante lorsque le tube est vide.
On peut pour cela modifier le flag O_NDELAY avec l'option F_SETFL de la primitive fcntl( ).

Remarque: La fonction fstat( ) vue pour les fichiers permet de connaître le volume des données présentes dans un pipe local.
La fermeture d'un pipe local intervient normalement lorsque les deux processus ont fermé tous les descripteurs qu'ils possédaient sur le pipe avec close( ):

Syntaxe: close (int desc);

Ils doivent tous deux effectuer un close(desc[0]) et un close(desc[1]).

A défaut le pipe est automatique démantelé à la fin du dernier processus l'utilisant.

Remarque: On peut aussi être attentif au signal SIGPIPE qui est émis par le kernel vers tout processus qui écrit dans un pipe alors qu'il n'y a plus de processus lecteur.

3) Les pipes nommés:

Un pipe nommé permet d'échanger des données entre deux processus quelconques à travers un fichier spécial de type fifo crée pour la circonstance dans le système de fichiers.

La création du pipe a lieu à l'appel de mknod( ):

Syntaxe: int mknod(char *pipename, int mode, int dev);

Cette primitive sert à créer tous les objets du système de fichier et est appelée par des fonctions
telles que creat( ), mkdir( ) ...
On l'utilise ici directement avec comme paramètres le nom complet du tube fifo à créer, le type de fichier S_IFIFO et les droits d'accès définis dans mode à l'aide des constantes habituelles (cf § Manipulations de fichiers à ce sujet).
Ici le paramètre dev ne sert à rien et sera mis à zéro.

Une fois crée le pipe nommé, on peut y accéder avec open( ) de la façon habituelle:

Syntaxe: int open(char *pipename, int flag);

Où flag est l'une des constantes O_RDONLY, O_WRONLY, O_RDWR selon les modalités
d'accès souhaitées.
Le flag O_NDELAY permet d'avoir une lecture non bloquante lorsque le tube est vide.

Remarques: Lorsque le flag O_NDELAY n'est pas levé, le premier des deux processus demandant l'ouverture restera bloqué jusqu'à l'ouverture du pipe par le second processus. Il y a donc par ce biais une synchronisation des deux processus désirant communiquer.

Si le flag O_NDELAY est levé, on doit impérativement commencer par l'ouverture en lecture, sans quoi une ouverture en écriture seule provoquera une erreur. Une alternative consiste à ouvrir le tube en lecture et écriture de part et d'autre.

Les opérations de lecture et écriture dans un pipe nommé passent encore par les fonctions read( ) et write( ).

La fermeture est toujours du ressort de close( ) qui doit être appelée par les deux processus.

Le fichier fifo n'est pas supprimé après la fermeture; on fera pour cela un appel à unlink( ) (cf § Manipulations de fichiers)

Remarque: Les fonctions stat( ) et fstat( ) permettent d'obtenir des renseignements sur un pipe nommé, notamment son état de remplissage.

Signalons enfin que sous System V Release 4 une fonction spécifique a été introduite pour la création des pipes nommés; il s'agit de mkfifo( ):

Syntaxe: int mkfifo(char *pipename, int mode);

Elle remplace alors l'appel à mknod( ) et on lui communique seulement le nom complet du pipe à créer et les droits d'accès.

VII) Les sockets.

1) Description:

On a vu que des processus peuvent communiquer sur une même machine grâce aux mécanismes d'IPC (Internal Process Communication).
Le mécanisme des sockets (c'est à dire prises) permet quant à lui de faire communiquer des processus s'exécutant sur des machines différentes, à travers un réseau.

Les sockets ont été originellement développés sous UNIX BSD, mais leur succès a conduit UNIX System V à les adopter rapidement.
Nous n'étudierons dans ce chapitre l'utilisation des sockets que dans le cadre de la famille de protocoles TCP/IP.

Concrètement un socket est un descripteur comparable au handle obtenu lors de l'ouverture d'un fichier; il est obtenu par appel à la fonction socket(), et permet de référencer un canal de communication réseau bidirectionnel entre deux machines utilisant les protocoles de la famille TCP/IP.

En général l'une des machines se comporte comme un serveur, ayant des services à offrir, et l'autre comme un client, souhaitant bénéficier de ces services: ainsi lors d'une connexion Telnet d'une machine UNIX sur une autre, la seconde exécute en permanence un logiciel serveur Telnet qui est à l'écoute des demandes de connexion sur le port TCP Telnet.

La bibliothèque de fonctions **socket** permet de réaliser à la fois des logiciels clients et des logiciels serveurs; il faut pour l'utiliser:

- Inclure les headers <sys/types.h>, <sys/socket.h>, <netdb.h> et <netinet/in.h>
- Linker la librairie socket au programme, sur certains systèmes.

On utilisera pour cela un makefile approprié.

Les sockets permettent aussi de faire communiquer des processus s'exécutant sur une même machine en utilisant de part et d'autre l'adresse IP de loopback localhost= 127.0.0.1 qui fait toujours référence à la machine elle même.
Cela permet entre autre de faire des essais avec une seule machine !

2) Les fonctions de manipulation de sockets:

a) Pour un processus client:

Un socket est alloué via la fonction socket():

Syntaxe: int socket(short famille, int type, int protocole);

On doit lui indiquer la famille de protocoles utilisée, ici AF_INET pour les protocoles internet TCP/IP.

Le type de socket alloué est spécifié par la variable type:

type= SOCK_STREAM	pour un socket TCP
type= SOCK_DGRAM	pour un socket UDP

Le paramètre protocole est nul pour cette famille de protocoles.
La fonction renvoie un numéro de socket en cas de succès et -1 en cas d'erreur.

Exemple: sock= socket(AF_INET, SOCK_STREAM, 0);

Un tel appel prépare un socket mais n'établit pas encore la moindre liaison car il faut préciser la machine avec laquelle on souhaite communiquer et les numéros de port TCP ou UDP à utiliser.
La variable sock servira aux manipulations ultérieures du socket.

L'extrémité locale d'un socket TCP ou UDP ainsi crée pour un processus client ne nécessite pas d'initialisation car l'adresse IP de la machine locale est gérée par les couches inférieures et le numéro de port local est attribué automatiquement.

Par contre il faut initialiser l'extrémité distante du socket et, s'il s'agit d'un socket TCP réaliser une connexion, ce qui incombe à la fonction connect():

Syntaxe: int connect(int sock, struct sockaddr_in *remote, int lgr);

On indique à connect() l'adresse IP de la machine distante avec laquelle on souhaite communiquer et le numéro de port TCP ou UDP à utiliser sur cette machine, en fonction du service recherché, à l'aide d'une structure du type suivant dont on passe l'adresse et la taille en paramètres:

```
struct sockaddr_in
        {
        short sin_family;               /* AF_INET */
        unsigned short sin_port;        /* numéro de port dans l'ordre normalisé */
        struct in_addr sin_addr;        /* voir ci-dessous */
        }

struct in_addr
        {
        unsigned long s_addr;           /* adresse IP sur 4 octets dans l'ordre usuel */
        }
```

La taille de la structure utilisée doit être lgr= sizeof(struct sockaddr_in).

Le champ sin_port est renseigné soit avec un numéro de port public de valeur supérieure à 1024, soit avec un numéro de port réservé à un service obtenu à l'aide de la fonction getservbyname() étudiée plus loin.

Le numéro de port doit être converti à l'ordre normalisé réseau.

La fonction connect() retourne -1 si la connexion n'a pu être établie dans le cas d'un socket TCP. Pour ce qui est des sockets UDP, l'appel à connect() est facultatif et permet simplement de ne pas avoir à préciser les coordonnées de la machine distante lors des échanges futurs....

💣 Attention: Les valeurs numériques sur 16 ou 32 bits destinées à être émises sur le réseau doivent être codées dans l'ordre normalisé réseau c'est à dire octets de poids fort en tête. Il existe des fonctions de conversion de shorts et de longs pour passer de l'ordre normalisé à l'ordre machine.

Remarque: Il existe des fonctions de conversion d'adresses IP sur 4 octets en notations décimales pointées et vice versa — Voir plus loin.

Ensuite les opérations d'émission et de réception de données sur un socket se font de diverses façons:

Syntaxe:

```
int write( int sock, char *tampon, int nb );
int send( int sock, char *tampon, int nb, int flag );
int sendto( int sock, char *tampon, int nb, int flag,
            struct sockaddr_in *remote, int lgr );
```

Les deux premières fonctions, write() et send() ne sont utilisables que pour un socket TCP ou un socket UDP après appel à connect(); la dernière, sendto() est destinée essentiellement aux socket de type UDP pour lesquels il n'a pas été fait appel à connect().

Toutes les trois se chargent d'émettre sur le réseau les nb octets de données contenus dans le tampon; il est à noter que seules les données du datagramme sont gérées par le programme, toute l'encapsulation TCP ou UDP et IP nécessaire est prise en charge par le stack TCP/IP sur lequel s'appuie la librairie socket.
Les fonctions send() et sendto() acceptent un flag que nous n'étudierons pas permettant d'émettre par exemple des données urgentes hors bande...
Pour des données hors bande (Out of band) sur un socket TCP, flag= MSG_OOB.

Syntaxe:

```
int read( int sock, char *tampon, int nb );
int recv( int sock, char *tampon, int nb, int flag );
int recvfrom( int sock, char *tampon, int nb, int flag,
              struct sockaddr_in *remote, int *lgr );
```

Les fonctions read() et recv() ne sont utilisables que pour lire les données reçues

sur un socket TCP ou un socket UDP après appel à connect(), tandis que recvfrom() est destinée essentiellement aux sockets UDP pour lesquels il n'y a pas eu de connect().

Le paramètre flag permet par exemple de détecter l'arrivée de données hors bande à traiter d'urgence...
Pour accéder aux données hors bande sur un socket TCP, flag= MSG_OOB.

Ces fonctions retournent le nombre d'octets effectivement reçus et -1 en cas d'erreur.
La lecture est par défaut bloquante jusqu'à réception de données.

Pour mettre fin à la connexion et fermer le socket rien de tel qu'un appel à close():

Syntaxe: close(int sock);

b) Pour un processus serveur:

S'agissant de réaliser un processus serveur associé à un port TCP ou UDP, on doit reprendre les opérations mentionnées ci-dessus, à cela près que l'appel à connect() qui est spécifique aux processus clients doit être remplacé par des appels aux fonctions bind(), listen() puis accept():

Syntaxe: int bind(int sock, struct sockaddr_in *local, int lgr);

On lui communique le socket concerné, l'adresse d'une structure de type prédéfini sockaddr_in identifiant l'adresse IP de la machine locale et le port TCP ou UDP à utiliser localement (c'est à dire le port auquel parviendront plus tard les demandes de connexion et les données entrant sur le socket), et la longueur en octets de cette structure déjà rencontrée.
La fonction bind() retourne -1 en cas d'anomalie.

Nous verrons plus loin comment obtenir l'adresse IP de la machine locale !

Syntaxe: listen(int sock, int nombre);
int accept(int sock, struct sockaddr_in *client, int *lgr);

Le rôle de la fonction listen() est de définir le nombre de requêtes entrantes qui devront éventuellement être maintenues en attente de traitement sur le socket sock; les requêtes excédentaires seront détruites.

La fonction accept() place le processus en attente d'une demande de connexion TCP ou d'une requête UDP sur le port associé au socket sock.
Cette fonction est bloquante et elle ne retourne que lorsqu'une requête arrive effectivement; un nouveau socket est alors retourné par la fonction, qui est une copie du socket d'origine sock, mais qui est connecté à la machine émettrice de cette requête.
Au retour la structure sockaddr_in passée par adresse est complétée avec l'adresse IP et le port de l'émetteur.

Le paramètre lgr passé par adresse indique la taille de la structure sockaddr_in lors de l'appel.

<u>Exemple:</u>

```
int sock, sk;
struct sockaddr_in client;
...
listen( sock, 5 );          /* mémoriser jusqu'à 5 requêtes entrantes */
sk= accept( sock, &client, sizeof(client) );
...
```

Dans cet exemple le socket sock initialisé par socket() et bind() est un socket générique appelé socket de rendez-vous: il sert à accepter les requêtes entrantes, mais ne sera lui-même
jamais connecté au client car ce rôle est dévolu au nouveau socket sk crée pour la circonstance.

Le processus serveur peut traiter les requêtes de deux façons:

Soit il exécute lui-même le traitement nécessaire puis ferme le socket sk ou le mémorise et reboucle sur accept(): c'est envisageable lorsque le traitement est simple, par exemple s'il s'agit d'un service UDP de base tel que l'écho UDP...

Soit il réalise un fork() et délègue le traitement au processus fils ainsi crée; dans ce cas le fils hérite automatiquement des sockets sock et sk; le père ferme immédiatement sk de façon à pouvoir le réutiliser en rebouclant sur accept() tandis que le fils ferme sock qui lui est inutile et travaille avec sk jusqu'à la fin du traitement ou jusqu'à la déconnexion...

Dans ce second cas le processus père doit surveiller la terminaison de tous ses enfants afin de ne pas forker au delà des limites raisonnables.
La déconnexion d'un client TCP peut être décelée lorsque read() ou recv(), normalement bloquantes jusqu'à l'arrivée de données, retournent avec une valeur nulle: cela signifie que le client s'est déconnecté et que le processus serveur peut libérer le socket.

3) Les noms de machines et les adresses IP:

Pour utiliser les fonctions recensées ci-dessus il faut connaître les adresses IP des machines locale et distante qui doivent communiquer.

La bibliothèque socket met à notre disposition des fonctions qui permettent de traiter les noms de machines internet et de les traduire en adresses IP, sous réserve de disposer bien sûr d'un serveur de noms de machines ou à défaut d'une base de données telle que /etc/hosts.

Les fonctions les plus utiles sont les suivantes:

Syntaxe:
```
int gethostname( char *nom, int taille );
int getdomainname( char *domaine, int taille) ;

struct hostent * gethostbyname( char *nom );
struct hostent * gethostbyaddr( char * adresse, int lgr, int type );
```

La fonction gethostname() permet au processus d'accéder au nom internet de la machine sur laquelle il s'exécute; au retour la chaîne nom dont l'adresse et la taille sont passés en paramètre contient le nom internet complet de la machine sous la forme habituelle host.domain.country
La taille de nom peut être égale à la constante prédéfinie MAXHOSTNAMELEN.
En cas d'erreur la fonction retourne (int) -1.

De même getdomainname() offre de récupérer le nom du domaine auquel appartient la machine sous la forme domain.country

Une fois que l'on a le nom internet de la machine il est facile de déterminer son adresse IP grâce à gethostbyname(): on lui passe le nom de cette machine — ou plus généralement d'une machine quelconque du domaine internet — et elle se charge de consulter le serveur de nom ou à défaut la base de données locale pour déterminer son adresse IP et autres renseignements contenus dans la structure hostent pointée au retour:

```
struct hostent
	{
	char *h_name;		/* nom officiel de la machine host */
	char **h_aliases;	/* liste des alias de cette machine terminée par NULL */
	int h_addrtype; /* type des adresses qui suivent= AF_INET */
	int h_length;		/* longueur de ces adresses= 4 octets */
	char *h_addr;		/* adresse IP principale de la machine host sur 4 octets */
	char **h_addr_list;	/* liste de toutes les adresses IP de cette machine
				   terminée par NULL */
	}
```

En cas d'erreur ou d'impossibilité la fonction retourne un pointeur NULL;

De même gethostbyaddr() permet d'identifier complètement une machine à partir de sa seule adresse IP sous forme de 4 octets dans l'ordre usuel; le champ lgr indique la longueur de cette adresse et type= AF_INET son type.

Signalons encore quelques fonctions utiles:

Syntaxe:
```
int getsockname( int sock, struct sockaddr_in *local, int *lgr );
```

int getpeername(int sock, struct sockaddr_in *local, int *lgr);

struct servent * getservbyname(char *service, char *protocole);

Les fonctions getsockname() et getpeername() extraient d'un socket sock des précisions sur l'extrémité locale et l'extrémité distante de ce socket, à travers une structure sockaddr_in dont l'adresse et la taille sont passés en paramètres.
Elles retournent -1 en cas d'erreur.

Enfin getservbyname() détermine le numéro de port réservé associé à un service TCP ou UDP en consultant par exemple le fichier /etc/services.
Elle renvoie un pointeur NULL si le service n'existe pas.

Exemple: service_ptr= getservbyname(″ftp″, ″tcp″);
service_ptr= getservbyname(″echo″, ″udp″);

La structure pointée par service_ptr est la suivante:

```
struct servent
	{
	char *s_name;		/* nom officiel du service tel que ftp*/
	char **s_aliases;	/* liste des alias de ce service terminée par NULL */
	short s_port;		/* numéro de port dans l'ordre normalisé */
	char *s_proto;		/* protocole en dessous de ce service, tcp ou udp */
	}
```

Quelques services disponibles sous TCP ou UDP:

```
echo		7	/* service d'écho qui renvoie les données reçues */
discard		9	/* service corbeille qui détruit les données reçues */
ftp-data	20	/* service ftp, canal de données, sous tcp */
ftp		21	/* service ftp, canal de communication, sous tcp */
telnet		23	/* service telnet, sous tcp */
smtp		25	/* service de e-mail, sous tcp */
name		42	/* service de noms de machines */
whois		43	/* service d'identification d'utilisateurs */
tftp		69	/* service tftp, sous udp */
```

4) Fonctions de conversion:

L'ordre normalisé réseau impose la conversion des valeurs sur 16 ou 32 bits émises sur le réseau; les fonctions suivantes assurent les conversions dans les deux sens:

Syntaxe: short ntohs(short s);
short htons(short s);

long ntohl(long l);
long htonl(long l);

Les deux premières convertissent des valeurs 16 bits dans le sens network to host ou host to network et les autres font de même pour les valeurs 32 bits.

Un autre problème de conversion concerne les adresses IP dont la représentation interne est sur 4 octets (toujours dans l'ordre usuel) mais que l'on représente plus volontiers en notation décimale pointée; on dispose de ces fonctions pour passer de l'une à l'autre:

Syntaxe: unsigned long inet_addr(char * decimal);
unsigned long inet_network(char * decimal);

char * inet_ntoa(struct in_addr adresse);

La première, inet_addr() transforme une chaîne contenant une adresse IP en décimal pointé en unsigned long, c'est à dire en 4 octets d'adresse IP dans l'ordre adéquat.

De même inet_network() extrait d'une adresse IP en décimal pointé la partie réseau de cette adresse, sur 4 octets.

Dans le sens inverse, le but de inet_ntoa() est de traduire une adresse IP mémorisée dans une structure de type in_addr — Voir au début — en une chaîne décimale pointée. Pour que cette fonction fonctionne convenablement il est impératif que le paramètre transmis soit une structure in_addr et non une vulgaire chaîne de 4 octets.

VIII) Les outils de développement et de mise au point.

1) Le compilateur cc:

Le compilateur cc fonctionne en ligne de commande. Pour la compilation de programmes simples on lance directement cc:

Syntaxe: cc -o *execname sources.c*
cc -o *execname* -l*libname sources.c*

L'option -o permet de définir le nom de l'exécutable généré. A défaut il s'agit de a.out
L'option -l permet d'introduire le cas échéant une librairie externe pour le linkage.

Exemples: cc -o toto toto.c
cc -o prog -lcurses prog1.c prog2.c

Il ne faut pas mettre d'espace entre l et le nom de la librairie invoquée.

Remarque: Le compilateur cc fait appel pour le linkage au linker nommé ld.

2) Le gestionnaire de projets make:

Les projets plus complexes peuvent comporter:

- plusieurs sources en langage C *.c
- des headers *.h
- des sources en assembleur *.a
- des fichiers objets précompilés *.o
- des librairies externes *.lib

Le compilateur cc compile les sources *.h et *.c en modules objets *.o, il assemble les sources
*.a en modules objets *.o également.
Le linker ld, appelé par cc, linke les fichiers *.o et *.lib pour générer l'exécutable attendu.

Afin d'automatiser toutes ces tâches et de les optimiser, un gestionnaire de projets nommé make peut être utilisé; il s'appuie en général sur des fichiers de description de projets
du nom de **makefile**:

Syntaxe: make *execname*
make -f *mkfile execname*

Lorsqu'on lance make, celui-ci recherche dans le répertoire courant un fichier de projet makefile ou Makefile (ou *mkfile* si on utilise l'option -f).
Il s'agit d'un fichier texte contenant la description des différents constituants du projet

et les actions à réaliser pour obtenir l'exécutable *execname*.

Pour être précis, make est capable de générer des cibles à partir de sources:

- les cibles intermédiaires sont les fichiers objets *.o
- la cible finale est l'exécutable

Pour chaque cible répertoriée, make recherche quels sont les modules nécessaires à l'obtention de cette cible:

- fichiers sources *.h et *.c
- fichiers sources assembleurs *.a

Il recompile alors les modules qui ont été modifiés depuis le dernier appel et met les cibles à jour.

Enfin l'appel au linker réalise le linkage des différents éléments:

- modules objets cibles *.o
- librairies indiquées *.lib

Un makefile comprend donc plusieurs rubriques que nous allons analyser sur un exemple:

<u>Exemple de makefile:</u>

```
#         flags du compilateur

CFLAGS=     -O  -s  -I/usr/include  -I.

#         objets et librairies

FILEOBJS=   uslink.o timing.o protocol.o
LIBS=       -lcurses -lmenu

#         dépendances des modules objets

uslink.o:       uslink.c uslink.h

timing.o:       timing.c

protocol.o:     protocol.c uslink.h

#         production de l'exécutable

uslink:  $(FILEOBJS)
         @$(CC)  -o $@  $(FILEOBJS)  $(LIBS)
```

Explications:

On lance la commande make uslink↵
make examine donc le fichier makefile ci-dessus.
Les commentaires sont introduits par des # et les noms des variables sont en majuscules comme dans un script shell.

a) Rubrique flags:

CFLAGS= −O −s −I/usr/include −I.

On définit ici des options du compilateur cc à appliquer à chaque source à compiler:

−O	Optimisation de l'exécutable
−s	Supprimer la table des symboles dans l'exécutable
−I*repname*	Rechercher les headers pour inclusion dans le répertoire indiqué

b) Rubrique objets et librairies:

FILEOBJS= uslink.o timing.o protocol.o

FILEOBJS décrit la liste des modules objets à linker pour former l'exécutable.

LIBS=		-lcurses -lmenu

LIBS contient la liste des librairies externes nécessaires pour le linkage.

c) Rubrique dépendances:

```
uslink.o:	uslink.c uslink.h
timing.o:	timing.c
protocol.o:	protocol.c uslink.h
```

Indique pour chaque cible ses dépendances en fonction des fichiers sources.
Ainsi make reconstruit la cible uslink.o si uslink.c ou uslink.h a été modifié.

d) Rubrique production:

```
uslink:	$(FILEOBJS)
	@$(CC)  –o $@  $(FILEOBJS)  $(LIBS)
```

La première ligne indique quelles sont les cibles objets à reconstruire pour l’exécutable uslink.
Voir ci-dessus pour leur reconstruction.

Une fois reconstruites les cibles objets on peut passer à la production de l’exécutable.
La ligne suivante, **qui commence par une tabulation**, est considérée par make comme une commande shell à exécuter, après substitution des variables par leur contenu:

$(CC)	désigne le compilateur à utiliser c’est à dire cc par défaut
$@	désigne la cible en cours de production, c’est à dire ici uslink
$(FILEOBJS)	désigne la liste des modules objets vue plus haut
$(LIBS)	désigne la liste des librairies définie plus haut

make exécutera donc dans un sous shell la commande
cc -o uslink uslink.o timing.o protocol.o -lcurses -lmenu

ce qui produira en fait un appel au linker ld pour réaliser l’exécutable uslink.

L’@ au début de la commande fait que celle-ci ne sera pas affichée par make.

Remarques:	Une commande de production commence toujours par une **tabulation**.
On peut en placer plusieurs pour un même exécutable, et dans la mesure où ces commandes sont exécutées par un sous shell on peut y mettre des commandes shell telles qu’echo, rm...

3) La mise au point:

Le debugger associé au compilateur cc porte le nom de sdb, dbx ou debug selon les systèmes UNIX.

Il s'agit d'un debugger symbolique capable de travailler avec les sources *.c et *.a, ainsi qu'avec les fichiers core générés en cas de plantage.

Pour utiliser le debugger au mieux avec un exécutable, il est nécessaire d'ajouter des paramètres particuliers à la compilation (Consulter l'aide à ce sujet).

L'interface offerte par le debugger est assez fruste et on n'y fera appel qu'en cas de nécessité absolue !

A

accept( ) 105
addch( ) 92
addstr( ) 92
alarm( ) 88
architecture 5
attributs 94
attroff( ) 94
attron( ) 94
attrset( ) 94

B

bind( ) 105
bg 43
break 43
BSD 3

C

cal 10
cancel 29
case 40
cat 20
cc 110
cd 15
chdir 15
chdir( ) 72
chgrp 18
chmod 17
chown 18
clear 20
clear( ) 93

close( ) 64, 100, 105
closedir( ) 73
connect( ) 103
connexion 7
continue 43
core 85
couleurs 94
cp 16
creat( ) 64
curses 90

D

date 10
dbx 113
debug 113
delwin( ) 97
démon 34
descripteur 63
DIR 73
dirent 74
domaine 56
doscp 49

dosdir 49
dosformat 49
dosls 49
dosread 49
doswrite 49
doupdate( ) 97
drivers de périphériques 13
droits d'accès 13
dup( ) 81
dup2( ) 81

E

echo 23
echo( ) 93
édition 26
endwin( ) 92
entrée standard 25
environnement 22
exec...( ) 75
exit 8
exit( ) 79
export 24
expr 38

F

fclose( ) 67
fcntl( ) 66
fflush( ) 70
fg 34
fgetc( ) 69
fgets( ) 69
fifo 13, 98
FILE 67
find 18
fopen( ) 67
for 42
foreach 47
fork( ) 77
fprintf( ) 69
fputc( ) 69
fputs( ) 69
fread( ) 69
fscanf( ) 69
fseek( ) 68
fstat( ) 66, 99
ftell( ) 68
ftp 57
fwrite( ) 69

G

getch( ) 93
getcwd( ) 72
getdomainname( ) 106
getenv( ) 80
gethostbyaddr( ) 106
gethostbyname( ) 106
gethostname( ) 106
getpeername( ) 107
getpid( ) 80
getppid( ) 80
getservbyname( ) 107
getsockname( ) 107
getstr( ) 93
goto 47
grep 19

H

handle 63
has_colors( ) 94
héritage 32

home 22
HOME 22
htons( ) 108
htonl( ) 108

I

if 40, 45
inet_addr( ) 108
inet_network( ) 108
inet_ntoa( ) 108
init_pair( ) 95
initscr( ) 92
IP 55
IPC 98

K

keypad( ) 93, 97
kill 33
kill( ) 87

L

ld 110
lien 13
link( ) 71
listen( ) 105
ln 16
login 7
logout 8
lp 28
lpstat( ) 28
ls 13
lseek( ) 65

M

mail 59
man 20
mkdir 15
mkdir( ) 72
mkfifo( ) 101
mknod( ) 100
more 20

move( ) 92
MS-DOS 49
mv 16

N

newwin( ) 95
noecho( ) 93
nohup 34
ntohs( ) 108
ntohl( ) 108

O

open( ) 64, 100
orphelin 34

P

passwd 7
PATH 22
path 22
pause( ) 88
pid 31
ping 57
pipe 98
pipe( ) 99
ppid 31
printw( ) 92
processus 30, 75
ps 31
psignal( ) 88
putenv( ) 80
pwd 15

R

read( ) 65, 99, 104
readdir( ) 73
recv( ) 104
recvfrom( ) 104
redirection 24
refresh( ) 93
remove( ) 71, 72
rename( ) 71
repeat 46
répertoire 13
requête 28
réseau 55
rewind( ) 68
rlogin 56
rm 16
rmdir 15
rmdir( ) 72

root 7

S

scanw( ) 93
script shell 36
sdb 113
send( ) 104
sendto( ) 104
set 20
setenv 23
shell 20
signal 85
sleep( ) 88
socket 102
socket( ) 102
sort 21
sortie erreur standard 24
sortie standard 24
start_color( ) 94
stat( ) 71, 101
stdin 70
stdscr 92
stty 20
subwin( ) 96
switch...case 46
symlink( ) 71
System V 3
system( ) 81
système de fichiers 12

T

tâche de fond 33
tar 48
TCP/IP 56, 102
tee 25
telnet 56
TERM 22
term 22
termcap 52
terminal 51
terminfo 52, 90
test 39
tty 10
tube 98

U

umask 18
umask( ) 71
unlink( ) 71
until 42

V

vi 26

W

waddch( ) 97
waddstr( ) 97
wait 33
wait( ) 79
waitid( ) 79
waitpid( ) 79
wattroff( ) 97
wattron( ) 97
wattrset( ) 97
wclear( ) 97
wgetch( ) 97
wgetstr( ) 97
while 41, 46
who 10
wmove( ) 97
wnoutrefresh( ) 96
wprintw( ) 97
wrefresh( ) 97
write 65, 99, 104
wscanw( ) 97

X

X-window 4

Z

Zombie 34