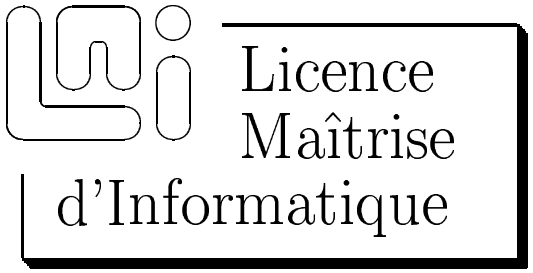
Université Bordeaux 1 Sciences Technologies

UTILISATION DES SYSTÈMES INFORMATIQUES

Introduction À L'Environnement De Développement Sous Unix

– Cinquième édition –

J.P.BRAQUELAIRE
(avec R. STRANDH)



– Octobre 2000 –

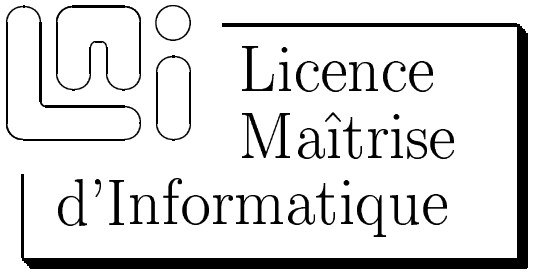
Université Bordeaux 1 Sciences Technologies

UTILISATION DES SYSTÈMES INFORMATIQUES

Introduction À L'Environnement De Développement Sous Unix

– Cinquième édition –

J.P.BRAQUELAIRE
(avec R. STRANDH)



– Octobre 2000 –

Avant propos

Ce texte est le support d'un cours d'initiation à l'environnement de programmation du système UNIX. Les premiers chapitres ne nécessitent aucun pré-requis. Les chapitres traitant de la programmation en shell nécessitent une connaissance des concepts de base de la programmation. L'accent est mis sur les logiciels du domaine public et du projet GNU qui constituent aujourd'hui la part la plus important de l'environnement de l'utilisateur et du développeur sous UNIX.

Table des matières

Chapitre 1

Présentation de l'environnement de travail sous Unix

Un environnement de travail informatique se compose de différents niveaux s'appuyant les uns sur les autres :

- la couche **matériel** (*hardware*) composée des ordinateurs, des périphériques et des réseaux d'interconnexion;
- la couche **système d'exploitation** (*operating system*) réalisant les tâches de base (gestion des fichiers, des communications...);
- la couche **application** qui est un ensemble de programmes permettant d'effectuer des traitements de plus haut niveau (édition et traitement de texte, programmation...).

Les deux dernières couches sont du logiciel (*software*).

L'environnement de travail du département d'informatique de l'université Bordeaux I est composé de stations et terminaux *X* tournant sous système UNIX et reliés à des **serveurs** au moyen d'un réseau ETHERNET.

1.1 Couche matériel

1.1.1 Configuration

Les machines de la licence d'informatique font partie d'un réseau local de machines, appelé `emi`,[1] sur lequel sont regroupées les différentes machines d'enseignement de l'UFR de mathématiques et informatique.

Le réseau considéré ici est un réseau ETHERNET. La vitesse optimale de ce type de réseau est de 100 Mégabits par seconde. Diverses unités sont interconnectées à travers ce réseau : des serveurs de disque, des stations de travail et terminaux X, et des imprimantes. Ce réseau est, comme de nombreux autres réseaux locaux de l'université Bordeaux I, connecté au réseau *REAUMUR* de l'université. Ce dernier assure la liaison avec le monde extérieur (voir figure 1.1).

Les postes de travail sont des postes de travail graphiques. Un poste de travail graphique est généralement composé :

- d'un clavier comportant des touches de caractère (lettres, chiffres, symboles), de touches de fonction, et de modificateurs qui sont des touches s'utilisant simultanément avec une autre touche : `CONTROL`, `SHIFT`, `META` ou `ALT`...
- d'un écran graphique dont la taille, la résolution, c'est-à-dire le nombre de pixels, et le nombre de couleurs affichables peut varier (par exemple 17 pouces, 1024×768 pixels, 256 couleurs);

1. `emi` est l'acronyme de *Enseignement de Mathématiques et Informatique.*

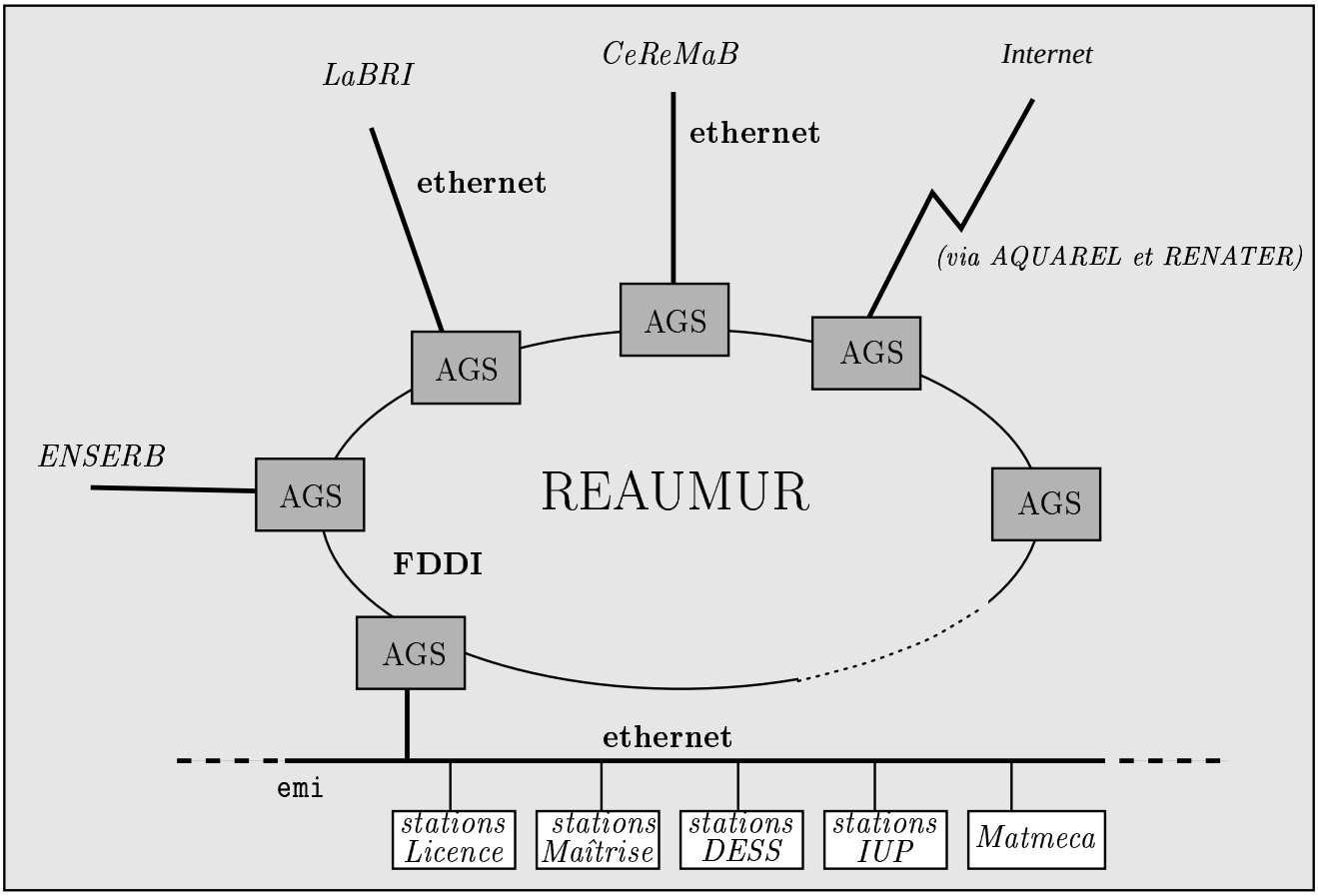


FIG. 1.1 – *Cette figure décrit l'organisation générale du réseau universitaire REAUMUR. Les réseaux locaux mentionnés sont divers réseaux en relation avec le laboratoire et le département d'informatique. De nombreux autres réseaux locaux sont connectés à REAUMUR.*

– d'une souris, comportant généralement trois boutons (`LEFT`, `MIDDLE`, `RIGHT`).

Chaque poste de travail est connecté par le réseau local à une machine centrale appelée serveur et gérant des unités de disque. Toutes les données des utilisateurs sont regroupées sur les disques des différents serveurs. L'accès à des imprimantes et lecteurs de disquette est également possible par le réseau.

Certains postes de travail sont des **stations** de travail, c'est-à-dire des ordinateurs, pouvant faire exécuter des programmes. D'autres sont des terminaux graphiques attachés à une station ou à un serveur. Ces terminaux sont capables de gérer localement des fenêtres et des opérations graphiques simples. Le système de gestion de fenêtres qu'ils implémentent est le système de fenêtrage *X*, ou *X11*, qui est le système de fenêtrage standard associé à UNIX. De tels terminaux sont des ordinateurs restreints, spécialisés dans la gestion du système *X*. Ils sont appelés des **terminaux X**, ou **TX**.

1.1.2 Événements et codage des caractères

Un **événements** est le résultat d'une action sur le clavier : frappe d'une lettre, d'une flèche... ou sur la souris : déplacement, enfoncement d'un bouton. Certains événements sont gérés localement sur le poste de travail, comme par exemple un déplacement de la souris provoquant celui du **curseur** sur l'écran. D'autres sont envoyés au système d'exploitation, qui s'exécute sur un ordinateur distant dans le cas d'un TX.

Les caractères sont des événements particuliers codés par des valeurs numériques appelées codes ASCII (*American Standard Code for Information Interchange*). Il s'agit de la norme de codage de caractères la plus largement utilisée.

Un premier ensemble de codes de base utilise les entiers compris entre 0 et 127. Il est listé sur la table 1.1. La plupart de ces caractères sont des caractères affichables : lettres, chiffres, ponctuation, espaces, parenthésages, etc. D'autres caractères, ceux dont le code est inférieur à 32 et celui de code 127, sont appelés des **caractères de contrôle**.

Chaque caractère de contrôle porte le nom d'un caractère affichable qui lui est associé. À chaque caractère de contrôle figurant sur la première colonne de la table 1.1 correspond le caractère affichable figurant sur la même ligne dans la troisième colonne. Par exemple, le caractère de code 1 est le caractère *CONTROL-A*, noté `C-a` ou `^A`. Le caractère `DEL` de code 127 est parfois noté `C-?`.

Chaque caractère de contrôle est par défaut associé à un traitement de base, comme par exemple :

– `C-m` (ou RETURN), `C-j` (ou LINEFEED) : validation d'une ligne entrée au clavier;
– `C-[` (ou `ESC`) : caractère d'échappement (ESCAPE);
– `C-i` (ou `TAB`) : insertion d'une tabulation;
– `C-h` (ou BACKSPACE) : déplacement du curseur vers la gauche;
– `DEL` : suppression du caractère à gauche du curseur;
– `C-s` et `C-q` : contrôle de flux;
– `C-l` : saut de page sur une imprimante.

Le plus grand code de la table 1.1 est l'entier 127. Il s'écrit 1111111 en base 2. Les codes ASCII sont par conséquent codables sur 7 chiffres binaires, encore appelés **bits**. L'unité de mémoire sur une machine étant traditionnellement l'**octet**, à savoir un mot de taille 8 bits, il est possible de coder 128 codes supplémentaires sur un octet. Ces codes sont les valeurs de 128 à 255. On appellera **méta-caractères**) ces caractères qui étendent l'ensemble des codes ASCII. Chacun porte le nom du caractère ASCII standard obtenu en retranchant 128 à son code. Par exemple le caractère de code 128 est `Meta-Control-@`, noté `M-C-@` (ou `C-M-@`), et celui de code 188 est `M-C-<`.

Les méta-caractères sont notamment utilisés pour coder les caractères des alphabets Européens : caractères accentués, ç, £, ß, ñ, Å, ¿, etc. Sur les micro-ordinateurs, ces codes peuvent également être utilisés pour coder des caractères graphiques. Le codage d'autres ensembles de caractères, arabe ou japonais par exemple, nécessite d'étendre encore ce codage.

Organisation de la table des codes ASCII							
Car.	Code	Car.	Code	Car.	Code	Car.	Code
^@	*0*	␣	*32*	@	*64*	‘	*96*
^A	*1*	!	*33*	A	*65*	a	*97*
^B	*2*	"	*34*	B	*66*	b	*98*
^C	*3*	#	*35*	C	*67*	c	*99*
^D	*4*	$	*36*	D	*68*	d	*100*
^E	*5*	%	*37*	E	*69*	e	*101*
^F	*6*	&	*38*	F	*70*	f	*102*
^G	*7*	’	*39*	G	*71*	g	*103*
^H	*8*	(	*40*	H	*72*	h	*104*
^I	*9*	)	*41*	I	*73*	i	*105*
^J	*10*	*	*42*	J	*74*	j	*106*
^K	*11*	+	*43*	K	*75*	k	*107*
^L	*12*	,	*44*	L	*76*	l	*108*
^M	*13*	-	*45*	M	*77*	m	*109*
^N	*14*	.	*46*	N	*78*	n	*110*
^O	*15*	/	*47*	O	*79*	o	*111*
^P	*16*	0	*48*	P	*80*	p	*112*
^Q	*17*	1	*49*	Q	*81*	q	*113*
^R	*18*	2	*50*	R	*82*	r	*114*
^S	*19*	3	*51*	S	*83*	s	*115*
^T	*20*	4	*52*	T	*84*	t	*116*
^U	*21*	5	*53*	U	*85*	u	*117*
^V	*22*	6	*54*	V	*86*	v	*118*
^W	*23*	7	*55*	W	*87*	w	*119*
^X	*24*	8	*56*	X	*88*	x	*120*
^Y	*25*	9	*57*	Y	*89*	y	*121*
^Z	*26*	:	*58*	Z	*90*	z	*122*
^[	*27*	;	*59*	[	*91*	{	*123*
^\	*28*	<	*60*	\	*92*	|	*124*
^]	*29*	=	*61*	]	*93*	}	*125*
^^	*30*	>	*62*	^	*94*	~	*126*
^_	*31*	?	*63*	_	*95*	DEL	*127*

TAB. 1.1 – *Table des codes ASCII*

1.2 Couche système

1.2.1 Le réseau

Plusieurs machines connectées à travers un réseau informatique doivent, pour se comprendre, respecter diverses conventions dans la façon d'échanger des messages. Un tel ensemble de conventions est appelé un **protocole de communication**. Le réseau considéré ici, comme la plupart des réseaux de machines UNIX, utilise le protocole TCP/IP. Ce réseau est à l'origine du réseau mondial INTERNET et à la base de son considérable succès.

Parmi les applications les plus intéressantes implémentées, citons :

- la connexion à distance, via le protocole TELNET et les commandes `telnet` et `rlogin`;
- le transfert de fichiers via le protocole FTP et la commande de même nom;
- le partage de fichier du système de fichiers réparti NFS distribué par *Sun Microsystems Incorporated*;
- le courrier électronique (protocole SMTP).
- le Web (WWW)

Chaque machine connectée à un réseau local (serveur, station de travail, terminal X, imprimante...) possède un nom qui permet de l'identifier de manière unique sur ce réseau. Le réseau local possède lui-même un nom qui permet de l'identifier parmi les autres réseaux réseaux locaux d'un réseau plus grand qui les relie. Il existe ainsi une hiérarchie à plusieurs niveau permettant d'identifier de manière unique chaque machine sur l'ensemble du réseau INTERNET.

Par exemple, le terminal X de nom `powell` est une station du réseau local `emi`, lui-même inclus dans le réseau `u-bordeaux` reliant différents sites universitaires bordelais. Ce réseau fait partie du réseau national `fr`. Le nom complet du terminal X `powell` s'écrit :

```
powell.emi.u-bordeaux.fr
```

Cependant, à l'intérieur du réseau `emi` le seul nom de la machine, c'est-à-dire `powell`, est suffisant pour l'identifier.

1.2.2 Le système d'exploitation

Un système d'exploitation est un programme complexe réalisant les fonctionnalités de base indispensables à tous les utilisateurs. Il existe plusieurs systèmes d'exploitation : UNIX, VMS, VM, etc. Sur les micro-ordinateurs tournent également des systèmes d'exploitation plus simples comme MS/DOS, ou MACOS. L'environnement considéré ici est le système UNIX.

1.2.2.1 Le système UNIX

Le projet UNIX a été initialisé par Ken Thompson, au laboratoire *Bell* de AT&T aux USA, à la suite du retrait de *Bell Labs* du projet MULTICS[2]. La date *officielle* du démarrage d'UNIX est fin 1969. Le 25*e* anniversaire du système a d'ailleurs été célébré durant le quatrième trimestre 1994. Il s'agissait au départ d'un projet de recherche, développé sur un PDP-7, par Ken Thompson et Dennis Ritchie. Le langage C a été développé à partir de 1971 par Dennis Ritchie et Brian Kernighan, et UNIX récrit en C en 1973. La première version officielle, UNIXV7, date de 1978. Elle a engendré deux grandes familles:

- BSD: version réalisée par l'université américaine de Berkeley;
- SYSTEM V: version réalisée par Bell Labs.

2. Le projet MULTICS était un projet ambitieux de réalisation de système d'exploitation qui a beaucoup contribué à faire évoluer les systèmes d'exploitation. Le nom d'UNIX est jeu de mot à partir du nom MULTICS.

Actuellement sont menées différentes tentatives de normalisation, dont la norme POSIX qui se situe à l'intersection de ces deux familles.

Voici quelques caractéristiques du système UNIX:

- *Système temps partagé*: exécution concurrente de processus, ordonnancement.
- *Système multi-utilisateurs*: comptes utilisateurs, connexion ou *login*, identification des ressources, environnement personnalisé.
- *Partage des ressources*: arborescence globale de fichiers, mécanismes de protections.
- *Gestion de communications*: entre processus, entre utilisateurs (`write`, `talk`, `mail`, *news*, *WWW*).
- Homogénéité, simplicité, portabilité, extensibilité.

1.2.2.2 Organisation générale du système UNIX

Le schéma de la figure 1.2 est traditionnellement utilisé pour représenter l'organisation du système UNIX. La partie centrale, le **noyau**, constitue le système proprement dit. Il a en charge:

- la gestion du système de fichiers, des processus, de la mémoire, des communications et des entrées sorties;
- le traitement d'événements extérieurs provenant des périphériques, les **interruptions**.

Un ensemble de commandes de base est fourni avec le système. Il est considérablement étendu au moyen de très nombreux logiciels libres et de logiciels du domaine public. Ces logiciels, souvent très puissants et très bien documentés constituent de plus en plus sous UNIX la base de l'environnement de travail de l'informaticien. Les noms figurant autour du noyau sur la figure 1.2 sont des **commandes**: interprètes de commandes, compilateurs, éditeurs de texte, gestion des connexions, commandes du système, programmes utilisateurs, etc.

Le système de fichiers est organisé de façon arborescente en répertoires et fichiers. De plus, toute ressource physique (imprimantes, disques, postes de travail...) est associée à un pseudo-fichier appelé **fichier spécial**. Grâce à cette organisation, les entrées-sorties sont homogènes. Leur mise en oeuvre est toujours la même, quel que soit la nature réelle du périphérique. On parle pour cela d'*entrées-sorties généralisées*. Chaque utilisateur possède une arborescence personnelle qui est une partie de l'arborescence générale. La racine de cette arborescence est son **répertoire d'accueil** *home directory*.

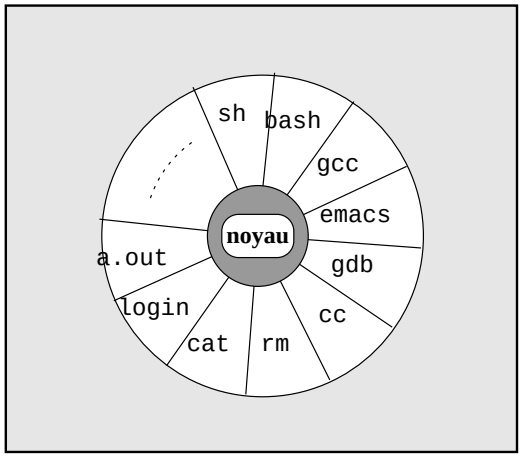


FIG. 1.2 – *Organisation du système UNIX*

1.2.2.3 Les interprètes de commandes

Un interprète de commandes est un programme itérant indéfiniment la lecture d'une ligne de commande au clavier et la mise en oeuvre de la (ou des) action(s) décrite(s) par cette ligne. Dans

certains cas l'interprète de commandes peut réaliser lui même ces actions, en utilisant éventuellement des services du noyau UNIX. Il s'agit de **commandes internes** à l'interprète. Dans la plupart des cas, ces actions sont réalisées par des commandes. De manière générale, l'interprète de commandes sert d'interface entre l'utilisateur et le système d'exploitation.

Sous UNIX, un interprète de commandes est une commande banalisée. Il en existe plusieurs, certains livrés en standard avec le système, et d'autres disponibles sur le réseau internet sous la forme de logiciels libres ou domaine public. Tous les interprètes de commandes UNIX sont basés sur les mêmes principaux généraux. Ils appartiennent à une même famille et sont désignés sous le terme générique de *shells*. La version de base commune à tous les systèmes UNIX est `sh`. Il en existe plusieurs extensions, dont l'interprète `bash` offrant une interface utilisateur très agréable et puissante.

Le fonctionnement général d'un shell est le suivant:

1) Lecture d'une **ligne de commande**:
   a) Affichage d'un message appelé *prompt* indiquant que l'interprète est prêt à lire une ligne de commande. Par défaut ce message est formé du caractère `$` suivi d'une espace.
   b) Saisie et édition de la ligne.
   c) Validation de la ligne.
2) Analyse et traitement de la ligne lue:
   a) Découpage en unités syntaxiques.
   b) Remplacement de *méta-caractères*.
3) Exécution de la (ou des) commande(s) correspondante(s).
   a) Traitement des redirections d'entrées-sorties
   b) Traitement de la commande dans le cas d'une commande interne, ou recherches et lancement des commandes à faire exécuter dans l'autre cas.

1.2.2.4 Les commandes

Les commandes standard du système peuvent varier d'une version d'UNIX à une autre. Cependant les commandes les plus utilisées se retrouvent sur toutes les versions. L'ensemble des commandes disponibles sur le réseau est très vaste. Ces commandes sont généralement fournies avec le source du programme. Cela permet d'étendre considérablement les possibilités d'UNIX. Ces extensions peuvent varier fortement d'un site à un autre. Cependant, lorsqu'une commande fait défaut sur un site, il est toujours possible et souvent relativement facile de l'y installer.

Pour lancer une commande, il suffit de taper son nom, et ce qu'il s'agisse d'une commande standard du système, d'une extension, voire même sous certaines conditions d'une commande développée par l'utilisateur lui-même (voir l'utilisation de la variable `PATH`, page 48. Les éventuels arguments sont tapés directement à la suite de la commande.

Chaque commande lancée donne naissance à un **processus**, qui est une occurrence de cette commande en machine. Il peut y avoir à un moment donné plusieurs processus différents provenant de l'exécution de la même commande. Voici un exemple de quelques exécutions de commandes.

```
$
$ whoami
modele
$
$ pwd
/users/modele
$
$ date
Wed Sep 13 13:50:28 MET DST 1995
$
$ cal 10 1995
```

```
October 1995
S  M Tu  W Th  F  S
1  2  3  4  5  6  7
8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31

$
```

La commande standard `ps` donne la liste des processus en cours d'exécution. L'interprète de commandes étant une commande banalisée, son exécution s'effectue sous la forme d'un processus, comme pour n'importe quelle autre commande. Sur l'exemple suivant, les deux processus listés par l'exécution de `ps` sont :

- l'interprète de commandes *bash*;
- la commande `ps` elle même.

```
$ ps
  PID TT   STAT  TIME COMMAND
  908 p2   S     0:10  (bash)
24218 p2   1     0:00 ps
$
```

Par défaut, seuls les processus de l'utilisateur qui lance `ps` sont listés. Il existe d'autres processus qui sont des processus créés au lancement du système et éventuellement des processus créés par d'autres utilisateurs.

1.2.3 L'interface X11

L'interface avec le poste graphique, c'est-à-dire l'ensemble clavier/souris/écran, est gérée par un programme particulier appelé **serveur X**.[3] Dans le cas d'une station, le serveur X est un processus UNIX. Dans le cas d'un TX, le serveur X est directement implémenté sur un processeur local au terminal.

Un programme qui utilise les fonctionnalités de X est appelé **client X**. La grande puissance de X11 est d'offrir la possibilité d'utiliser un poste graphique à travers le réseau en permettant à un client d'interagir avec un serveur distant. C'est sur cette fonctionnalité qu'est basé le fonctionnement d'un terminal X.

Dans la terminologie X, un poste graphique est appelé un **display**. Un *display* est identifié par un nom de machine suivi d'un identificateur qui est généralement la chaîne de caractères `:0.0`. Par exemple, le serveur X du terminal X de nom `murnau`, est par défaut identifié par la chaîne `murnau:0.0`.

On dispose sous un shell d'un ensemble de variables appelées **variables d'environnement** (voir également section 2.4.3 page 22 et section 5.4). Ces variables peuvent être exportées automatiquement à tous les programmes lancés par le shell. Ce mécanisme peut être utilisé pour indiquer à un client le serveur qu'il doit utiliser. On place pour cela le nom du client dans une variable spéciale dont le nom est `DISPLAY` typographié en majuscules. Dans l'exemple précédent, avec le shell `bash`, on écrit simplement :

```
$ DISPLAY=murnau:0.0
```

3. Attention à ne pas confondre un «*serveur de fichiers*», qui est un ordinateur central dédié à la gestion de disques, avec un «*serveur X*» qui un programme spécialisé implémentant le système de gestion de fenêtres X11.

```
$ export DISPLAY
$
```

1.3 La couche applications

1.3.1 L'environnement standard

L'environnement logiciel standard d'UNIX est relativement riche, mais déjà assez ancien. Divers langages de programmation sont disponibles : le langage C (pas toujours la version normalisée ISO/ANSI), le langage FORTRAN, un *assembleur*, les langages interprétés SH, CSH, AWK, etc. Sont également disponibles de façon moins standard, soit dans des distributions spécifiques à certains constructeurs soit en domaine public, des langages comme SCHEME, LISP, PROLOG, ADA, EIFFEL, C++, etc.

Les principaux outils standard d'aide à la programmation sont :

- le débogueur langage machine `adb`,
- le débogueur symbolique `dbx`,
- des générateurs d'application comme `make` (voir 4.3),
- les générateurs de programmes `lex` et `yacc`,
- des outils rudimentaires d'analyse comme `prof`,
- différentes bibliothèques, comme la bibliothèque de programmation de terminaux `termcap`.

Il existe de nombreuses commandes de base de manipulation de données (tri, fusion, recherche...) facilement combinables ensemble et permettant de construire très facilement des traitement complexes. Cet aspect *boite à outils* à été une des raisons du succès remarquable d'UNIX pendant les années 80.

Citons enfin les éditeurs de texte `ed` et `vi` qui ne présentent plus guère d'intérêt pour les utilisateurs[4] et les outils de traitement de texte `nroff` et `troff` utilisés pour mettre en page les pages du manuel.

1.3.2 L'environnement X

1.3.2.1 Ressources

Le comportement du serveur est paramétrable au moyen d'un ensemble de **ressources**. Il s'agit de données gérées par le serveur et codant différentes informations comme les polices de caractères disponibles, le paramétrage d'une fenêtre (couleur de fond, police courante, épaisseur d'un trait...), la forme et la vitesse du curseur, etc.

Cet environnement est complètement dynamique, et modifiable au moyen de la commande `xrdb`. Il est courant de l'initialiser au moyen d'un fichier de configuration de nom `.Xdefaults` qu'on fournit à la commande `xrdb` au moment de la connexion.

Cette requête est en général elle même placée dans le fichier d'initialisation `.xsession` qui est automatiquement pris en compte lors de la connexion. Cela permet à chaque utilisateur de posséder ses propres fichiers d'initialisation, et par conséquent de personnaliser son environnement.

4. Ces outils sont parfois utilisés par les administrateurs du système lorsqu'ils installent un nouveau système ou qu'ils réparent un système abîmé. Dans ce cas en effet, il est fréquent qu'on ne dispose que d'un environnement restreint dont ne fait pas partie un outil de plus haut niveau comme *GNU Emacs*.

1.3.2.2 Le gestionnaire de fenêtres

Un gestionnaire de fenêtre X est un client particulier qui assure l'interface entre le serveur X et l'utilisateur[5]. Les plus courants sont `twm`, `ctwm`, `mwm`, `awm`. Les fonctionnalités de base sont communes à tous ces gestionnaires :

- Gestion des fenêtres : création, suppression, empilement (*pop/push*), déplacements, modification de taille, etc.
- Iconification : ouverture et fermeture de fenêtres, dessin et placement des icônes, etc.
- Gestion de menus.

Ces gestionnaires lisent au moment de leur lancement un fichier d'initialisation (*start-up*) permettant de paramétrer et d'adapter leur comportement (`.twmrc`, `.mwmrc` par exemple).

On peut voir sur la figure 1.3 un exemple d'écran sous X avec différentes fenêtres et icônes. Sur cet exemple, le gestionnaire utilisé est `mwm`.

1.3.2.3 Émulateur de terminal

Un terminal simple est un ensemble clavier/écran connecté par deux *canaux* à un programme appelé **contrôleur de terminaux**. Pour chaque caractère entré au clavier, le contrôleur renvoie un **écho** de ce caractère sur l'écran. Il mémorise chaque caractère reçu dans une zone mémoire appelée un **buffer** jusqu'à ce qu'un programme effectue une lecture. Sur les terminaux vidéo non graphiques, également appelés terminaux ASCII, ces deux canaux correspondent à une paire de fils (voir figure 1.4).

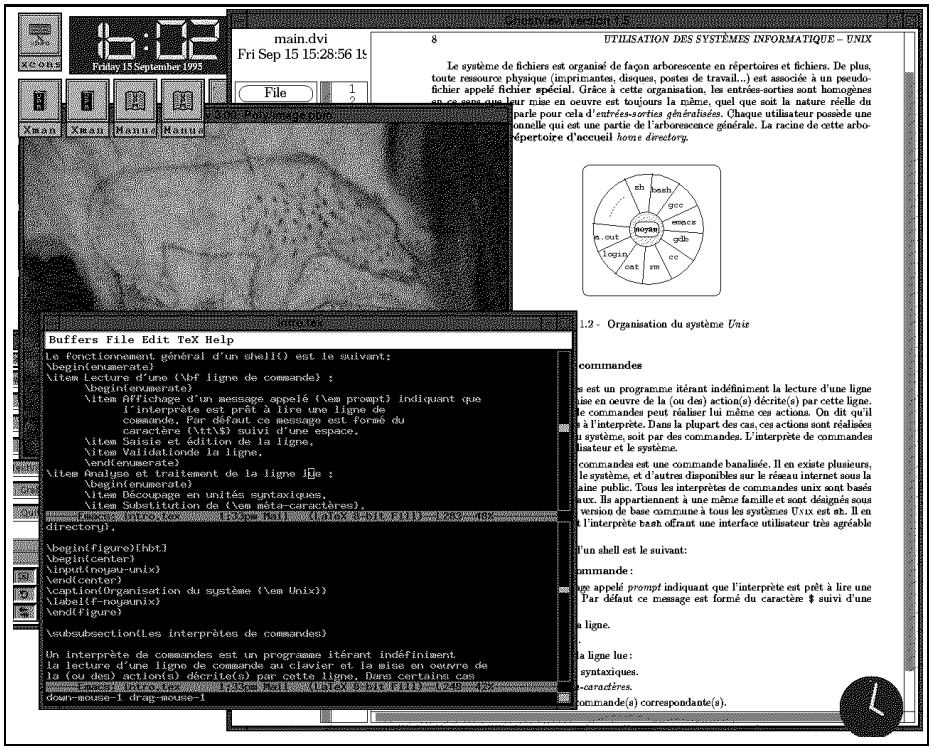


FIG. 1.3 – *Exemple d'écran d'une station de travail sous X*

Ce mode d'interaction était sous UNIX le seul possible jusqu'à la banalisation des terminaux X et des stations graphiques. Bien que cette nouvelle technologie permettent des fonctionnalités beaucoup plus puissante, comme l'implémentation **d'hypertextes multimédia**, le fonctionnement de très nombreux programmes, comme par exemple les interprètes de commandes, est encore régi par ce mode d'interaction de base. Pour interagir avec ces programmes, on utilise sous X un logiciel appelé

5. On notera l'analogie qu'il existe avec un shell qui est une commande UNIX particulière assurant l'interface avec le noyau.

émulateur de terminal, qui simule par un logiciel le comportement d'un terminal vidéo ASCII. L'émulateur de terminal le plus courant est `xterm`.

1.3.2.4 Divers clients

De très nombreux clients sont disponibles réalisant les fonctionnalités les plus diverses. Citons par exemple les horloges `xclock`, `oclock` et `dclock`, l'indicateur de courrier `xbiff`, le lecteur de nouvelles `xrn`, l'interface WWW `netscape`, la calculatrice `xcalc`, le manuel d'utilisation `xman`, l'outil de manipulation d'images `xv`, `gimp` etc. Sur la figure 1.3 sont ouvertes différentes fenêtres associées à *GNU Emacs*, ghostview, xv, dclock et oclock.

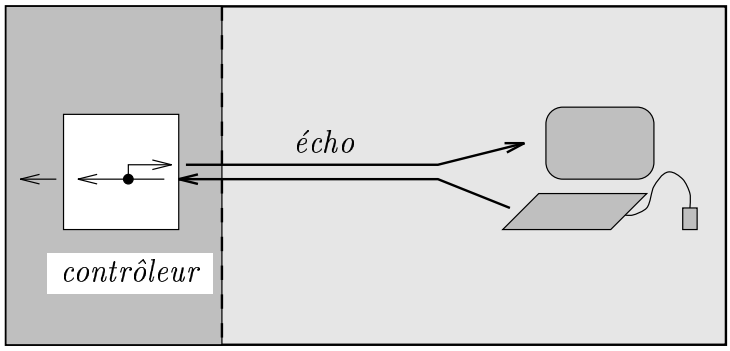


FIG. 1.4 – *Organisation d'un terminal ASCII*

1.3.3 L'environnement GNU

Le projet GNU, dirigé par Richard Stallman, est un projet de développement d'un système compatible UNIX, gratuit, et dont tous les sources sont accessibles. Pour développer ce système, différents outils ont été réalisés comme l'éditeur *GNU Emacs* ou le compilateur `gcc`. Le système GNU est disponible avec deux noyaux différents : *HURD* ou LINUX. Les outils GNU sont aussi disponibles sur d'autres systèmes UNIX (comme Solaris ou HP) et même sur certains systèmes non UNIX comme Windows NT. La qualité et le grand nombre des outils distribués sous le label GNU en font un environnement extrêmement intéressant. Les principaux outils étudiés ici, l'éditeur *GNU Emacs*, les compilateurs C, C++ et OBJECTIVE C regroupés dans la commande `gcc`, le débogueur symbolique `gdb`, l'interprète de commandes `bash`... sont issus de ce projet.

Signalons en particulier le projet LINUX de développement d'un noyau UNIX librement distribuable tournant sur micro-ordinateurs compatibles PC. Ce projet est associé au projet GNU pour proposer un système complet, le noyau étant le noyau LINUX et les commandes celles du projet GNU. Cet ensemble constitue à l'heure actuelle l'outil le plus intéressant pour travailler sous UNIX avec un micro-ordinateur.

1.3.3.1 L'éditeur : *GNU Emacs*

Le logiciel *GNU Emacs* est un **éditeur de texte**, c'est-à-dire un programme permettant de saisir et de modifier un fichier texte. Emacs est une famille d'éditeurs de texte remontant à environ 1975, très répandue sur de nombreux systèmes. La version la plus récente de *GNU Emacs* est 20.3. Un autre membre de la famille est *XEmacs* réalisé par Lucid Inc, et basé sur GNU Emacs.

À partir de la version 19, *GNU Emacs* est pleinement interfacé avec X. Voici quelques unes de ses caractéristiques.

- Interfaçage X11.
- Documentation hypertexte et mécanismes dynamiques d'aide.

- Multi-fenêtres, multi-buffers, multi-fichiers (voir figure 1.5), répartis dans des écrans (`frame`) correspondant à des fenêtres X.
- Fonctionnement en *mode insertion* et activation de commandes au moyen de clés.
- Fonctionnalités classiques : déplacements dans le fichier, insertion/suppression de lignes, de caractères, de paragraphes, de blocs..., recherches et positionnements, etc.
- Interfaçage avec le système : parcours de répertoire, lecture du courrier, interaction avec les *shells*, compilations, etc.
- Environnement extensible, programmable en LISP.
- Bibliothèque de modes spécialisés.

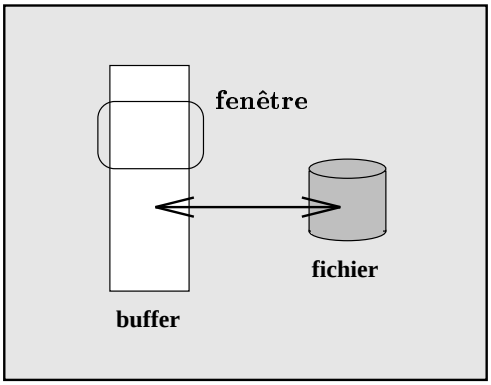


FIG. 1.5 – *Chaque fichier ouvert par GNU Emacs est recopié dans un buffer et affiché sur l'écran dans une fenêtre. Il est possible d'ouvrir plusieurs fenêtres sur un même fichier. Le nombre de buffers n'est pas à priori limité.*

1.3.3.2 L'interprète de commandes : bash

Il s'agit d'un shell standard offrant de très nombreuses possibilités d'édition des commandes, rappelant les fonctionnalités de base de *GNU Emacs*.

1.3.3.3 Compilateur et débogueur : gcc et gdb

Le compilateur `gcc` est un compilateur C implémentant la norme C ISO/ANSI. Il intègre également un compilateur C++ et un compilateur OBJECTIVE C, qui sont deux langages à objets définis au-dessus du langage C.

Il y a plusieurs intérêts à utiliser le compilateur `gcc` : implémentation de la norme ISO/ANSI, clarté des messages d'erreur, rapidité du compilateur, efficacité du code généré lors de l'utilisation de l'optimiseur de code.

Le débogueur symbolique `gdb` est un outil indispensable au débogage rapide d'une application. Il permet notamment de suivre instruction par instruction l'exécution d'un programme compilé et d'interagir sur le processus directement au niveau du source.

Les deux outils, `gcc` et `gdb` acquièrent toute leur puissance lorsqu'il sont utilisés directement sous *GNU Emacs*.

1.3.3.4 Divers

Mentionnons simplement pour référence quelques uns des autres outils du projet GNU :

- *Ghostscript*, `ghostview`, `gspreview` : visualisation de fichiers POSTSCRIPT.

- `gnumake` : générateur d'application.
- `bison` : générateur de programme.
- `gnuplot` : visualisation de fonctions mathématiques.
- `groff` : traitement de texte des pages de manuel UNIX.
- `GIMP` : logiciel de traitment d'images.
- `Lilypond` : logiciel de mise en page de partitions musicales.
- `Octave` : logiciel similaire à Matlab.
- etc.

1.3.4 Le traitement de texte TEX/LATEX

Le traitement de texte TEX développé par Donald Knuth et disponible en domaine public est actuellement l'outil de traitement de texte le plus puissant du point de vue des possibilités typographiques. Il respecte les usages et le savoir-faire séculaire des typographes et des composeurs.

Intégrant un langage de programmation, il est entièrement reprogrammable et donc extrêmement extensible. Il existe sur les serveurs internet une multitude de fichiers appelés **fichiers de style** permettant d'adapter et d'étendre TEX.

L'utilisation de TEX est facilitée par une couche définie au-dessus de celui-ci appelée LATEX, adapté à la composition de textes livres, de rapports, d'articles, de documents techniques, etc. Ce polycopié a été rédigé en utilisant le mode `book` de LATEX. Un texte en LATEX (ou en TEX) est un fichier ASCII contenant du texte normal et des commandes commençant par un caractère `\`. Voici par exemple le code en LATEX des premières lignes de cette section :

```
\subsection{Le traitement de texte \TeX{}/\LaTeX}

Le traitement de texte \TeX{} développé par Donald Knuth et disponible
en domaine public est actuellement l'outil de traitement de texte le
plus puissant du point de vue des possibilités typographiques. Il
respecte les usages et le savoir-faire séculaire des typographes et
des composeurs.

Intégrant un langage de programmation, il est entièrement
reprogrammable et donc extrèmement extensible. Il existe sur les
serveurs internet une multitude de fichiers appelés {\bf fichiers de
style} permettant d'adapter et d'étendre \TeX.
```

Les commandes indiquent des actions à effectuer sur le texte, comme le sectionnement ou un changement de police. On peut notamment voir sur cet exemple que la numérotation des sections et les césures[6] sont effectuées automatiquement par TEX.

Un fichier TEX (resp. LATEX) est compilé au moyen de la commande `tex` (resp. `latex`). Le résultat est un fichier graphique appelé fichier *dvi* (pour *DeVice Independent*). Il est possible de convertir ce fichier en fichier PostScript imprimable sur une imprimante laser au moyen de la commande domaine public `dvips`. Un fichier *dvi* peut être visualisé sous X au moyen du client `xdvi`. Le fichier PostScript peut être visualisé au moyen de ghostscript (voir 1.3.3.4).

Signalons également la possibilité de construire des figures au moyen du client X domaine public `xfig`. Ces figures sont facilement intégrable à un source LATEX au moyen de la commande domaine public `fig2dev`.

6. Les césures, c'est-à-dire le découpage des mots est paramétrage selon la langue dans laquelle on écrit. Par exemple les règles de césure diffèrent en français et en anglais.

1.4 Compte utilisateur et protection

Au démarrage du système, chaque poste de travail est prêt à accepter la **connexion** d'un utilisateur. Sur l'écran est affiché un message de la forme

```
login:
```

Le système connaît la liste des utilisateurs autorisés à se connecter sur la machine. Chaque utilisateur est identifié par un nom appelé son *nom de login*. Lorsqu'un utilisateur est défini sur une machine, on dit qu'il possède un **compte** sur cette machine.

Pour des raisons de sécurité, tous les comptes ouverts sur une machine doivent impérativement être protégés par un **mot de passe**. La raison principale est d'interdire l'accès de la machine aux personnes n'ayant pas de compte. Cela permet également de restreindre, voire de supprimer, l'accès aux données d'un utilisateur pour les autres utilisateurs.

Chaque nouvel utilisateur reçoit généralement un mot de passe initialisé par l'administrateur du système. Lors de la première connexion, la première chose à faire et de définir un nouveau mot de passe.

Il est impératif que chaque mot de passe ne soit connu que de son seul propriétaire

Ce mot de passe doit être suffisamment *robuste* pour résister à des tentatives d'effractions. Deux critères important de robustesse sont la longueur et la complexité du mot.

- Longueur : tout mot de passe devrait comporter 8 caractères qui est le maximum pris en compte.
- Complexité : tout mot de passe devrait contenir des caractères variés, par exemple en mêlant des lettres majuscules et minuscules et des caractères spéciaux (virgules, espaces, accents...).

Il faut éviter **absolument** les mots de passe simples à deviner : son prénom, sa date de naissance, un mot en rapport avec son activité favorite, etc. D'autre part, il existe des programmes de *craquage* de compte utilisant des dictionnaires de mots. Il faut donc éviter tout mot figurant dans un dictionnaire, ou en rapport avec un thème ou un sujet à la mode.

La sécurité des comptes est une chose très importante sur des machines reliées au réseau Internet. Les tentatives d'effraction et de piratage venant de l'extérieur ne sont pas rares. Il est important de bien comprendre qu'un compte facilement fracturable peut être utilisé par des *pirates* comme tremplin pour effectuer de nouvelles effractions. Dans ce cas le propriétaire du compte peut être incriminé, s'il a commis des imprudences, comme par exemple prêter son compte à une autre personne en lui divulguant son mot de passe.

Le mot de passe est défini ou modifié au moyen de la commande `passwd` sur une machine isolée et de la commande `yppasswd` sur une machine en réseau. Lors de la saisie du mot de passe, l'écho est supprimé de façon à ne pas laisser de trace de ce mot sur l'écran. Pour éviter les erreurs de frappe, la commande redemande une seconde fois ce mot de passe. Dans le cas d'un changement de mot de passe, l'ancien mot de passe est également demandé.

Chapitre 2

Introduction au système UNIX

2.1 Rôle du noyau

Le noyau est un programme chargé en mémoire centrale lors du lancement du système. Il demeure en mémoire jusqu'à l'arrêt du système : on dit qu'il est **résident**. Pour des raisons de sécurité, il est le seul à pouvoir accéder à la totalité de la mémoire, à la différence des processus utilisateur qui sont enfermés dans un espace mémoire propre. De cette façon, un processus ne peut agir directement sur les données d'un autre processus ou du noyau.

La communication entre un processus et le noyau se fait au moyen d'un mécanisme spécial appelé **appel système**. Il s'agit d'une requête envoyé au noyau, lui demandant soit de fournir une information sur l'état du système, soit d'effectuer une action sur le système (par exemple créer un fichier).

Chaque unité périphérique de la machine (consoles, disques, imprimantes...) est commandée par un **contrôleur** (contrôleur de terminaux, contrôleur de disques...) Un processus utilisateur ne peut accéder directement à un contrôleur de périphérique.

Certains modules du noyau, appelés les **pilotes de périphérique** ou *device drivers*, sont chargés de l'interface entre les processus et les contrôleurs. La communications avec les pilotes se fait également au moyen d'appels systèmes.

2.2 Les entrées-sorties

2.2.1 Flots d'entrées-sorties

Une **opération d'entrée-sortie** est généralement un transfert de données entre une zone mémoire et un fichier ou une unité périphérique. Par exemple un processus peut effectuer des entrées-sortie sur le terminal depuis lequel il a été lancé (lectures au clavier et écritures sur l'écran).

On appellera **flots d'entrée-sortie** d'un processus ses entrées-sorties actives. Plusieurs types de communications sont possibles :

processus	$\longrightarrow$	fichier
fichier	$\longrightarrow$	processus
processus	$\longrightarrow$	périphérique
périphérique	$\longrightarrow$	processus

Il est également possible d'établir un flot d'entrée-sortie entre deux processus :

processus	$\longrightarrow$	processus

2.2.2 Flots d'entrées-sorties standard

Un processus possède par défaut à sa création trois flots d'entrés-sorties prédéfinis appelés **flots d'entrées-sorties standard** (ou simplement *entrées-sorties standard*). Ce sont :

- `stdin` : **flot d'entrée standard**, sur lequel sont effectuées les opérations de lecture standard; il est par défaut associé au clavier du terminal;
- `stdout` : **flot de sortie standard**, sur lequel sont effectuées les opérations d'écriture standard; il est par défaut associé à l'écran du terminal;
- `stderr` : **flot de sortie erreur standard**, sur lequel sont effectués les affichages de messages d'erreur; il est par défaut associé à l'écran du terminal.

Le schéma de la figure 2.1 représente les flots d'entrées-sorties par défaut d'un processus.

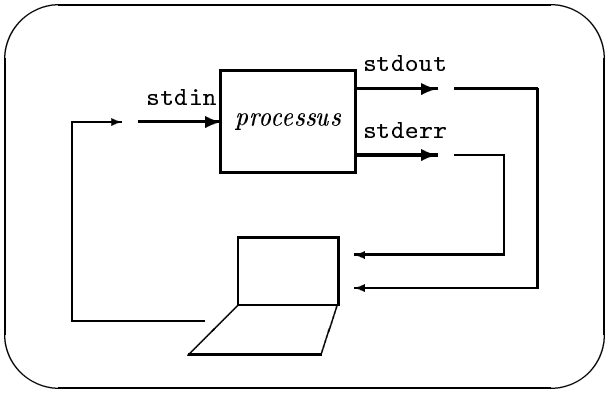


FIG. 2.1 – *Flots d'entrées-sorties standard*

2.2.3 Mécanismes de redirection

Il est possible de redéfinir les flots d'entrées-sorties standard en les associant à un nouveau périphérique ou à un fichier. On parle dans ce cas de **redirection** d'entrée-sortie.

Les redirections sont possibles directement depuis un *shell*. On distingue deux cas :

- d'un programme vers un fichier : par exemple, la commande

  ```
  $ ls -l >ls.l
  ```

  redirige le flot de sortie standard de `ls` dans un fichier de nom `ls.l` (Figure 2.2);

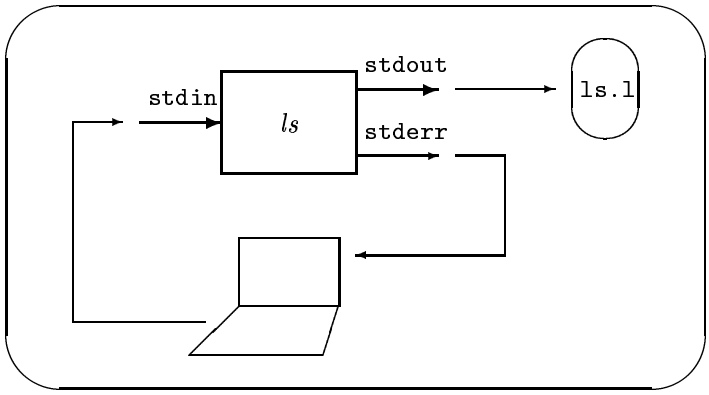


FIG. 2.2 – *Redirection de sortie standard*

– d'un programme vers un autre programme (**pipe** ou **tube**) : par exemple, la commande

```
$ ls -l | more
```

redirige la sortie standard de `ls` sur l'entrée standard de `more` (Figure 2.3).

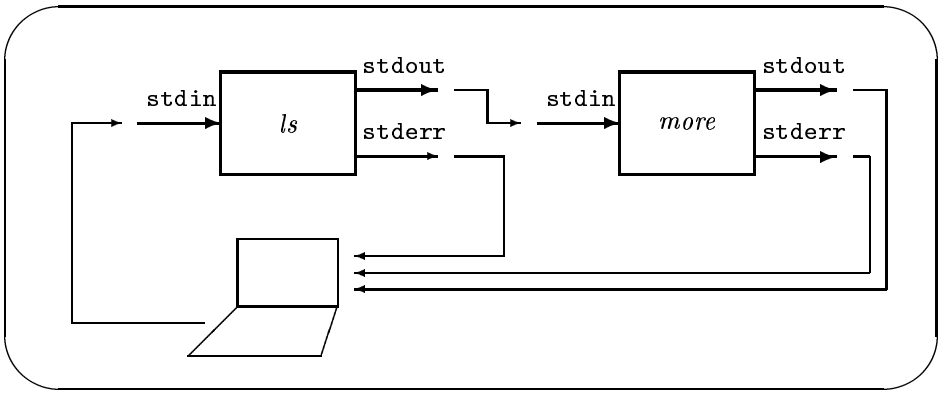


FIG. 2.3 – *Résultat d'un pipe*

La possibilité d'assembler des commandes au moyen de tubes à fait se multiplier les **filtres** sous UNIX. Un filtre est une commande traitant un flot de données lue sur son entrée standard et transmettant le résultat sur sa sortie standard. Les plus courants sont :

- `grep` : extraction de motifs
- `sort` : tri des lignes
- `tr` : remplacements de caractères (voir section 5.5.3 page 53)
- `sed` : remplacement de motifs (voir section 5.5.4 page 54)
- `awk` : langage de programmation permettant de réaliser des traitements complexes sur un flot
- `head` : extraction des premières lignes
- `tail` : extraction des dernières lignes

2.2.4 Fichiers spéciaux

Sous UNIX, le terme général de fichier peut désigner plusieurs types de fichiers :

- *fichiers réguliers* : fichiers contenant des données (textes, programme sources, commandes exécutables...);
- *répertoires* : fichiers contenant d'autres fichiers;
- *fichiers spéciaux* : fichiers fictifs permettant la liaison avec un pilote de périphérique.

Par exemple, `/dev/tty` désigne le terminal courant, et `dev/tty`***nn*** le terminal numéro *nn*. Du point de vue de l'utilisateur, un fichier spécial se comporte comme un fichier régulier. Par exemple, la commande

```
date >/dev/tty12
```

affiche la date sur le terminal numéro 12 (si l'accès en écriture est autorisé). On qualifie les entrées-sorties UNIX d'**entrées-sorties généralisées**.

2.3 Le système de fichiers

Le système de fichiers est un arbre dont la racine est notée `/`. Chaque processus possède un **répertoire courant** qui est sa position dans l'arbre du système de fichiers. Lors de la connexion, le répertoire courant du *shell* est le répertoire d'accueil (*home directory*). Il peut être modifié au moyen de la commande `cd`.

2.3.1 Description de l'arborescence standard

- répertoires de fichiers exécutables : `/bin`, `/usr/bin`, `/usr/local/bin`
- répertoires de bibliothèques : `/lib`, `/usr/lib`, `/usr/local/lib`
- pages de manuel (sur UnixBSD) : `/usr/man/man`*i* (pages sources), `/usr/man/cat`*i* (pages formatées)

  Le manuel Unix est composé de pages décrites dans un langage de traitement de textes spécifique à Unix. Deux commandes sont utilisées pour traiter le texte :
  - `nroff` qui prépare une sortie pour écran vidéo,
  - `troff` qui produit un résultat pour imprimante laser (voir figure 2.4).

  Lorsque l'on demande une page de manuel au moyen de la commande `man`, le message

  `Wait reformatting...`

  signifie que le page à afficher est calculée au moyen de `nroff`. Elle est ensuite stockée dans le répertoire des pages formatées afin d'éviter de la recalculer lors d'une prochaine consultation.
  Un gros avantage de cette organisation est que ce qui s'affiche à l'écran par `man` et ce qui est imprimé dans la documentation papier provient des mêmes fichiers source.

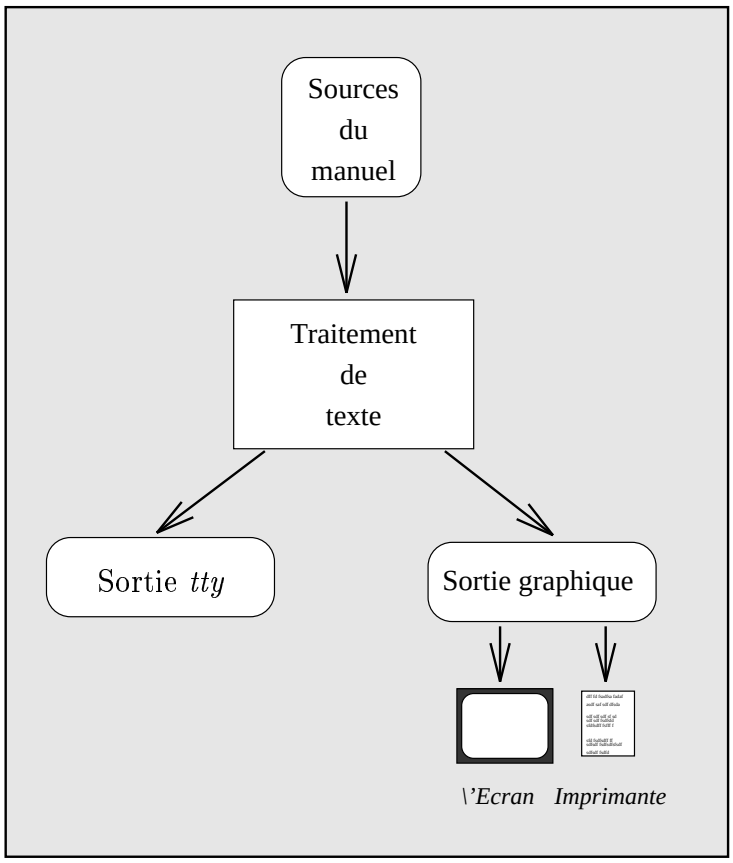


Fig. 2.4 – *Formatage des pages de manuel par nroff et troff.*

- Fichiers d'administration : `/etc`, `/adm`
  - `/etc/passwd` : chaque ligne de ce fichier définit un utilisateur

    `login:passwd:uid:gid:commentaire:répertoire d'accueil:shell`

    Chaque utilisateur est identifié par le système au moyen d'un numéro unique appelé son **uid** (*user identifier*). Chaque ligne du fichier `passwd` met en correspondance un nom de *login* avec un *uid*. Elle contient toutes les informations nécessaires à la connexion, y compris le mot de passe crypté.

  - `/etc/termcap` : base de description de terminaux
  - `/etc/rc` : initialisation du système
  - `/etc/utmp` : liste des utilisateurs connectés
  - ...
- divers : `/tmp`, `/dev`, `/users`

2.3.2 Chemin

2.3.2.1 Chemins absolus et relatifs

Un **chemin** identifie un fichier (fichier régulier, répertoire...) dans le système de fichiers. Il correspond à un déplacement dans l'arborescence des fichiers. De plus, dans chaque répertoire R, il existe deux noms réservés

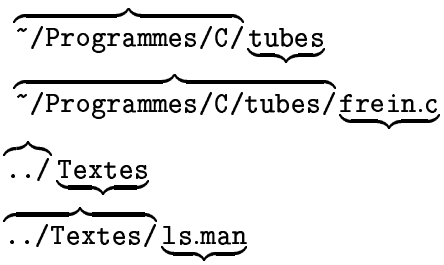
- `..` : référence au répertoire père de R
- `.` : référence au répertoire R

- **Chemin relatif**. Il exprime un déplacement par rapport à la position courante; par exemple, si le répertoire courant est `/usr/local`, les chemins
  1) `bin`
  2) `bin/emacs`
  3) `../lib`
  4) `../lib/libtermcap.a`

  sont des chemins de répertoires dans les cas 1) et 3), et des chemins de fichiers dans les deux autres.
- **Chemin absolu**. Il exprime un déplacement par rapport à la racine (indépendant de la position courante) :
  1) `/usr/local/lib`
  2) `~/emacs` (`~` est sous *bash* une abréviation pour le répertoire d'accueil).

2.3.2.2 Validité d'un chemin

Un chemin peut être décomposé en un **préfixe** et **nom de base**. Lorsque le chemin contient des `/`, le nom de base est le mot suivant le dernier `/`; le reste est le préfixe :



Le préfixe est toujours le chemin d'un répertoire; il détermine la position dans l'arborescence du répertoire ou du fichier identifié par le nom de base. Lorsque le chemin ne comporte pas de `/`, il se réduit au seul nom de base. Le préfixe est alors implicitement `.`, c'est-à-dire le répertoire courant.

Un chemin est **valide** seulement si son préfixe est le chemin d'un répertoire existant, que l'on appellera **répertoire de référence** du chemin.

2.3.3 Protections et accès

- Un fichier appartient :
  - à un utilisateur, qui est son propriétaire;
  - à un groupe (pas nécessairement celui de son propriétaire).
- Trois types d'accès sont définis :
  - pour un fichier : *lecture*, *écriture*, *exécution*;
  - pour un répertoire : *lecture*, *écriture*, *utilisation* (il s'agit de l'utilisation dans la partie préfixe d'un chemin).
- Le **mode d'accès** définit l'accès en $\left\{\begin{array}{l}\text{lecture}\\\text{écriture}\\\text{exécution}\end{array}\right.$ pour $\left\{\begin{array}{l}\text{le propriétaire}\\\text{les utilisateurs de même groupe}\\\text{les autres utilisateurs}\end{array}\right.$.

  Cela fait 9 informations élémentaires (3 champs de 3 flags), que l'on note

$$\underbrace{\texttt{rwx}}_{u}\underbrace{\texttt{rwx}}_{g}\underbrace{\texttt{rwx}}_{o}$$

  Le caractère `-` indique qu'un accès n'est pas autorisé :

```
rw-r--r--
rwxr-xr-x
```

2.3.4 Mécanismes de remplacement de chemins

Sous l'interprète de commandes, il est possible de spécifier des listes de chemins au moyen de *motifs*. Par exemple, le motif `*` représente tous les noms de fichiers du répertoire courant dont le nom ne commence pas par un point. De manière générale, dans un motif, `*` remplace une liste quelconque de caractères. Exemples :

```
echo /u*/l*
echo /users/*
```

Au moment du traitement de la commande, c'est-à-dire avant la phase d'exécution, *bash* remplace le motif par la liste des chemins lui correspondant, puis transmet les éléments de cette liste en argument à la commande. Il existe d'autres mécanismes de substitution (voir 5.3.4).

2.3.5 Principales commandes relatives aux fichiers

- `cd` : déplacement dans l'arborescence
- `pwd` : affichage du chemin du répertoire courant
- `ls` : listage du contenu de répertoires
- `cat` : listage du contenu de fichiers
- `more` : affichage du contenu
- `cp` : copie de fichier
- `mv` : déplacement de fichier
- `rm` : suppression de fichier
- `chmod` : changement des droits d'accès
- `chown` : changement du propriétaire d'un fichier
- `cmp` : comparaison de deux fichiers
- `diff` : listage des différences entre deux fichiers
- `file` : identification de fichiers
- `strings` : recherche de chaînes de caractère dans un exécutable

– `join` : jointure de deux fichiers

Citons également quelques filtres (voir page 2.2.3) pouvant aussi s'utiliser sur des fichiers :

– `grep` : recherche de motifs
– `head` : listage du début d'un fichier
– `tail` : listage de la fin d'un fichier
– `sort` : tri

2.4 Les processus

2.4.1 Création d'un processus

Un **processus** est une occurrence en mémoire d'un programme exécutable. Seul le noyau peut créer de nouveaux processus. Un processus peut demander au noyau le lancement d'un nouveau processus au moyen d'appels systèmes spécifiques. Un tel appel reçoit en paramètre

– le chemin du programme à charger;
– la liste éventuelle de ses arguments.

Le noyau teste si le processus demandeur a le droit d'exécuter ce programme : et si c'est le cas, effectue le lancement. Chaque nouveau processus reçoit un numéro unique appelé son **pid** (*process identifier*). Les numéros sont attribués par le noyau modulo un numéro maximal.

C'est de cette façon que les shells, par exemple *bash*, font exécuter les commandes entrées au clavier. En utilisant ces appels, il est possible de programmer de nouveaux interprètes de commandes.

La liste de tous les processus actifs à un instant donné est stockée en mémoire dans une table appelée la **table des processus**. Cette table est mise à jour par le noyau à chaque création ou suppression de processus.

2.4.2 État d'un processus

Un processus n'utilise pas la machine en permanence. En particulier, lorsqu'il demande un accès à une ressource physique, il peut être mis en attente, le temps que l'accès soit effectué.

On peut connaître l'état des processus en appelant la commande `ps` avec l'option `-l`. Les principaux états par lesquels passe un processus sont résumés sur la figure 2.5.

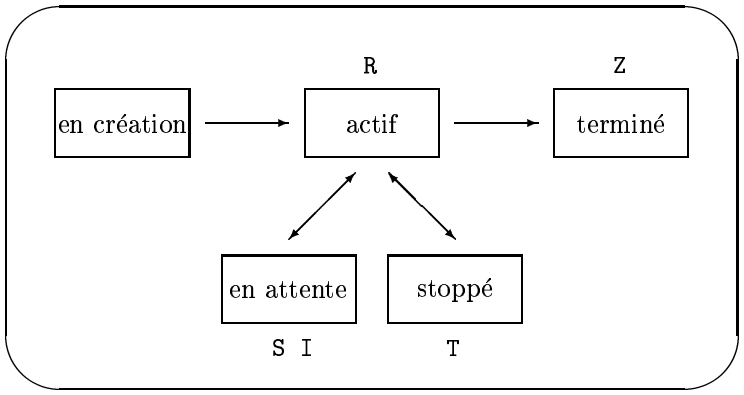


FIG. 2.5 – *États d'un processus*

2.4.3 Environnement d'un processus

2.4.3.1 Environnement système

Il s'agit des informations dont le noyau a besoin pour gérer l'ensemble des processus. Ces informations sont accessibles via la table des processus. Parmi les informations de l'environnement système d'un processus, on trouve :

- son *pid*,
- son état,
- le numéro de son propriétaire et de son groupe,
- le *pid* de son père, c'est-à-dire celui qui a provoqué sa création,
- son répertoire courant,
- son niveau de priorité,
- ses entrées et sorties standard,
- la partie de la mémoire où il est chargé,
- ...

La plupart de ces informations sont transmises du processus père au processus fils par recopie lors de la création du fils.

2.4.3.2 Données

On distingue

- les données proprement dite du programme;
- les arguments de lancement de la commande : il s'agit d'une liste de chaînes de caractères;
- l'**environnement utilisateur** : il s'agit d'une liste de *variables* dont le contenu (la valeur) est une chaîne de caractères. Parmi ces variables, certaines sont prédéfinies au moment du *login*, puis transmises par défaut de père en fils, comme par exemple :
  - `HOME` : le chemin absolu du répertoire d'accueil
  - `TERM` : le type du terminal vidéo
  - `PATH` : une liste de chemins de répertoires dans lesquels sont rangés des programmes exécutables.
  - `DISPLAY` : nom du serveur X.

  Les variables transmises aux processus fils sont appelées des variables **exportées**.
  Certaines de ces données sont dans une *pile* appelée la **pile d'exécution** du processus. On peut connaître la liste des variables d'environnement exportées par le *shell* `printenv`.

2.4.4 Contrôle des actions d'un processus

Au moyen des appels systèmes, tout processus peut demander la mise en oeuvre d'opérations

- sur son propre environnement système :
  - modifier sa priorité,
  - se mettre en attente d'un événement,
- sur des ressources physiques :
  - création ou suppression de fichiers,
  - activation d'entrés-sorties,
  - ...
- sur d'autres processus
  - communications (par exemple les *pipes*),

– modification de l'état (par exemple stoppage, ou terminaison),

– ...

Toutes ces opérations sont rigoureusement contrôlées par le noyau. On a déjà évoqué en 2.3.3 les contrôles et mécanismes de protections concernant les fichiers. De même, un processus ne peut interagir sur un autre processus que s'ils appartiennent tous les deux à un même utilisateur.

Il existe une seule exception à ce mécanisme. Il s'agit d'un utilisateur privilégié, appelé le *super-utilisateur*, qui est l'utilisateur d'*uid* zéro. En général, son nom de *login* est `root`.

Lorsqu'un processus appartient à cet utilisateur (on dit alors que le processus est super-utilisateur), le noyau n'effectue plus aucun contrôle sur les appels systèmes. Ce processus est tout puissant : il peut accéder à toutes les ressources physiques, interagir sur tous les processus, etc.

L'accès super-utilisateur, c'est-à-dire la connexion sous le nom de `root`, est réservée aux personnes qui administrent la machine.

2.4.5 Communication entre processus

Nous avons vu en 2.2.3 qu'il est possible de faire communiquer deux processus au moyen de pipes, en redirigeant le flot de sortie standard de l'un sur le flot d'entrée standard de l'autre.

Il est également possible de faire communiquer deux processus par signaux. Un **signal** est une information élémentaire, codée par un entier, qu'un processus peut envoyer à un autre. Le noyau peut également envoyer des signaux aux processus.

Lorsqu'un processus reçoit un signal, il suspend aussitôt le traitement en cours pour traiter le signal reçu. Par défaut, cela provoque sa terminaison, avec dans certain cas la production d'un fichier `core` qui est une copie de l'image mémoire du processus. Par exemple, une erreur dans un programme peut entrainer l'utilisation d'une adresse mémoire incorrecte et causer l'envoi par le noyau du signal `SIGBUS` qui provoque la terminaison du programme avec le message

```
Bus error - core dumped
```

Un processus peut aussi récupérer un signal afin d'effectuer un traitement spécifique à la place de la terminaison.

2.4.6 Principales commandes relatives aux processus

2.4.6.1 Job control

Un processus peut être stoppé pendant son exécution. Cela se fait par défaut en tapant `C-z` pendant l'exécution. Le processus stoppé reste présent dans le système. Il peut à tout moment être relancé au moyen de la commande `fg`. Il reprend dans ce cas son exécution normalement.

Par défaut, un processus s'exécute en **avant-plan** (*foreground*). Cela signifie qu'il possède le contrôle du terminal depuis lequel il a été lancé (par exemple une fenêtre `xterm`). Il est possible de détacher un processus afin de placer son exécution en **arrière-plan** (*background*). Dans ce cas, il ne peut plus faire de lecture au terminal. C'est le shell de ce terminal qui reprend la main. Il y a deux façons de placer un processus en arrière plan :

– terminer la commande de lancement par le caractère `&`,

– stopper le processus et le relancer au moyen de la commande interne `bg`.

On peut obtenir une liste de ses processus, avant-plan, arrière-plan et stoppés, au moyen de la commande interne `jobs` :

```
$ jobs
[1]   Running                 oclock -fg #f0b050 -bd #0b3709 &  (wd: ~)
[2]   Running                 dclock &  (wd: ~)
```

```
[4]   Stopped                 grep  (wd: ~)
[8]   Running                 xbiff &  (wd: ~/lmi/recrutement)
[9]   Running                 emacs &  (wd: ~/lmi/plaquette/organisation)
[12]   Running                 netscape ~/html/index.html &
[13]-  Running                 ghostview main.ps &
[14]+  Stopped                 latex main
$
```

Le caractère `+` indique le processus stoppé courant, c'est-à-dire celui qui est relancé par une utilisation des commandes `fg` ou `bg`. Le numéro entre crochets est utilisable pour désigner un autre processus. Par exemple `fg %13` rattache le processus `[13]`. Attention, cette numérotation des processus de l'utilisateur est interne à bash et ne correspond pas à l'identificateur de processus attribué par le système. Les identificateurs système (*pid*) peuvent être obtenus au moyen de la commande `ps` :

```
$ ps
  PID TT STAT  TIME COMMAND
 2207 p1 S     0:53 /usr/new/bash -login
 2239 p1 IW    2:36 oclock -fg #f0b050 -bd #0b3709
 2240 p1 IW    4:02 dclock
 2243 p1 T     0:01 grep
 2408 p1 S     5:12 xbiff
 2429 p1 IW   35:58 emacs
 8921 p1 IW    3:41 xfig
14773 p1 IW    4:35 netscape /usr/labri/achille/html/index.html
21269 p1 IW    0:13 ghostview main.ps
21273 p1 IW    0:03 gs -sDEVICE=x11 -dNOPAUSE -dQUIET -dSAFER -
21342 p1 T     0:00 latex main
21343 p1 R     0:00 ps
$
```

2.4.6.2 Commandes usuelles

Voici quelques autres commandes classiques de manipulation des processus.

- `kill` : envoi d'un signal à un processus (sous bash, on peut utiliser indifféremment le *pid* ou le numéro interne donné par la commande `jobs`)
- `who` : liste des utilisateurs connectés
- `w` : idem, mais avec d'avantage d'informations (BSD)
- `nice` : lancement d'un processus en faible priorité
- `time` : lancement d'un processus avec obtention de temps d'exécution
- `trace` : traçage de l'exécution d'un processus

2.4.7 Mécanismes de «boot» et de «login»

Lors du lancement du système, le noyau est tout d'abord chargé dans la partie de la mémoire qui lui est réservée. C'est l'étape de *bootstrap*, ou *boot*. Lorsque le chargement est terminé, le système Unix est opérationnel. Le noyau crée un premier processus, appelé processus initial, de nom `init` et de *pid* 1. Ce processus est chargé à partir du programme exécutable `/etc/init`.

Le processus `init` lance, pour chaque terminal à connecter, la commande `/etc/getty`, dont le rôle est d'attendre la demande de connexion d'un utilisateur, en affichant sur le terminal le message

```
login:
```

Lorsqu'un utilisateur tape son *nom de login*, `getty` lit ce nom, puis fait exécuter la commande `login` en lui transmettant le nom lu en argument.

Ce nouveau processus effectue les actions suivantes :

- recherche de la ligne définissant cet utilisateur dans `/etc/passwd` et extraction des informations (mot de passe crypté, numéro d'utilisateur...);
- lecture du mot de passe (avec suppression de l'écho);
- cryptage du mot lu et comparaison avec le mot crypté;
- si *ok*,
  - initialisation du propriétaire, groupe, et répertoire courant du processus;
  - initialisation des entrées-sorties standard;
  - lancement de l'interprète de commandes.
- sinon affichage de

```
Login incorrect
login:
```

L'interprète de commandes commence par lire un ou plusieurs fichiers d'initialisation, appelés fichiers *start-up*. Par exemple, `bash` lit les fichiers `.bash_profile` et `.bashrc` du répertoire d'accueil. Il affiche ensuite le prompt et attend la frappe d'une commande.

Exercices.

---

1. Mettre un mot de passe sur son compte.
2. Essayer les commandes `who`, `tty`, `date`, `cal`, `echo`.
3. Essayer `man man`.
4. Rechercher une commande qui nettoie l'écran.
5. Essayer les commandes d'édition de ligne de *bash*:
   - faire afficher le calendrier de l'année courante;
   - utiliser l'historique et l'édition de ligne pour faire successivement les calendriers des mois de janvier, de décembre, et de février de l'année courante.

   Essayer la recherche avec `C-r`.
6. Essayer les mécanismes de complétion de *bash* (`C-i`, `M-?`).
7. Dessiner l'arborescence du système de fichiers. Décrire les principaux répertoires du système UNIX.
8. Essayer ces différentes options de `ls` dans le répertoire d'accueil.
9. Essayer `more` ⟨*fichier*⟩; que remarque-t-on par rapport à `man`?
10. Essayer la commande `file` sur tous les fichiers d'un répertoire (par exemple `/etc/*`).
11. Essayer la commande `wc` sur un fichier, sur une liste de fichiers, sans fichier.
12. Essayer `echo ~`. Comment peut-on faire afficher le caractère `~`?
13. Essayer les commandes suivantes les motifs de remplacement de noms de fichiers. En particulier, essayer `echo */*`, `echo */??` à différents endroits de l'arborescence.
14. Créer un répertoire `tmp` et un répertoire `Projets` dans son répertoire d'accueil.
15. Essayer `rmdir ~` et commenter.
16. Donner des exemples de commandes effectuant des lectures et des écriture dans un répertoire.
17. Essayer `ls -l`, `ls -lg`. Regarder les différents accès de ses fichiers et de fichiers du système. Essayer diverses modifications par `chmod` vérifier le résultat au moyen de `ls -l`,
18. Expérimenter les modes d'accès d'un répertoire au moyen de `chmod`, `ls`, `cp` et `rm`. Vérifier que l'on peut avoir le droit de détruire un fichier sans avoir le droit de le lire. Dans quel cas peut-on détruire un fichier qui ne nous appartient pas?
19. Créer dans son répertoire `~/tmp` un fichier contenant du texte et enlever l'accès en lecture pour l'utilisateur. Essayer de lire au moyen de `cat`. Vérifier qu'un autre utilisateur peut encore le lire. Sous *GNU Emacs*, charger le fichier protégé en lecture. Stopper *GNU Emacs* et remettre l'accès en lecture. Recommencer les accès lecture.

20. Enlever l'accès en écriture pour l'utilisateur sur un fichier texte. Réveiller *GNU Emacs* et tenter de modifier le fichier: *GNU Emacs* a placé ce buffer en mode `read only` repérable par `--%%-` en début de ligne de mode. Ce mode est désactivable au moyen de la commande `toggle-read-only` liée à la clé `C-x C-q`. Désactiver ce mode, modifier le fichier et tenter de sauver. Stopper *GNU Emacs*, redonner l'accès en écriture et recommencer. Essayer ces manipulations sur un fichier d'un autre utilisateur.

21. Pourquoi `ls -l` ne permet pas de lister le mode d'accès d'un répertoire; rechercher au moyen de `man` la façon de faire.

Chapitre 3

Introduction à Emacs

3.1 Définitions et concepts de base

3.1.1 Jeu de caractères

GNU Emacs est capable de traiter n'importe quel caractère ASCII étendu, codé sur 8 bits. Les caractères de 0 à 127 sont ceux de la table ASCII standard; les caractères allant de 128 à 255 sont des *méta-caractères* (voir page 3). Par défaut, les caractères non affichables sont visualisés à l'écran de la façon suivante :

```
  0  ^@
  1  ^A
  2  ^B
     ⋮
 27  ^[
 28  ^\
 29  ^]
 30  ^^
 31  ^_
```

```
127  ^?
128  \200
129  \201
     ⋮
255  \377
```

Sous X, il est également possible d'activer l'option d'affichage graphique des caractères européen codés par des méta-caractères (voir également page 3). Cela se fait au moyen de la commande `standard-display-european` (voir section 3.2.1 pour l'invocation d'une commande *GNU Emacs* et le fichier `emacs.el` listé page 33 pour le positionnement par défaut de l'affichage européen).

Sur les postes de travail ne disposant pas de la touche *Méta*, on peut simuler sous *GNU Emacs* le caractère `M-c` en tapant la suite de deux caractères `ESC c`.

3.1.2 Buffer, fenêtres et écrans

Sous *GNU Emacs*, un **buffer** est une zone mémoire, indéfiniment extensible[1], dans laquelle sont rangés des caractères quelconques. *GNU Emacs* peut gérer simultanément un nombre quelconque de buffers. Cependant, il est raisonnable de ne pas saturer la mémoire par un trop grand nombre de buffers.

On appelle **point** l'endroit du buffer à partir duquel agissent les commandes d'édition. Cette position est située entre deux caractères. Le curseur est toujours situé sur le caractère qui suit le point.

GNU Emacs peut tourner sur un terminal simple, par exemple dans une fenêtre xterm, en en

1. (dans la mesure de la mémoire, centrale et/ou disque, physiquement disponible)

utilisant les possibilités de fenêtrage de X11. Afin d'éviter toute ambiguïté, nous appellerons **écran** ou *frame* une fenêtre X ouverte par *GNU Emacs*. Sur un terminal simple, *GNU Emacs* ne dispose que d'un seul écran. Sous X, Emacs peut ouvrir autant d'écrans que souhaité.

Un écran peut être divisé horizontalement ou verticalement en **fenêtres** de taille variable. Chaque fenêtre est obligatoirement associée à un buffer, dont le contenu est visualisé dans la fenêtre.

Chaque fenêtre est munie d'une **ligne de mode** décrivant son état courant (position courante du point, nom du buffer, mode courant, marques *buffer modifié*...).

Il existe en bas de l'écran une fenêtre associé à un buffer particulier : le **mini-buffer**. Cette fenêtre est par défaut réduite à une seule ligne. Elle est utilisé pour effectuer des opérations d'entrées-sorties.

3.1.3 Clé

Une clé est une suite de caractères interprétée comme un tout par *GNU Emacs*. La plupart des caractères sont des clés. Certains autres sont des caractères préfixes et ne constituent pas des clés à part entière. C'est par défaut le cas de

```
C-c
C-x
C-h
ESC
```

Le séquences préfixes ne sont pas limités à un seul caractère. Par exemple, par défaut la séquence

```
C-x 4
```

est un préfixe.

Les préfixes ne sont pas *câblés* dans *GNU Emacs*. Au contraire, il est à tout moment possible de redéfinir le statut d'un caractère, y compris de faire de n'importe quel caractère un caractère préfixe.

Il est possible d'insérer une clé dans le texte grâce à un mécanisme de *quotation*. On utilise pour cela la clé `C-q` liée à la commande `quoted-insert`. Par exemple, la suite de caractères `C-q C-d` insère le caractère `C-d` dans le buffer; il est visualisé par `^D`. De même, `C-q ESC v` insère les caractères `ESC` et `v`, et `C-q C-q` insère le caractère `C-q`.

3.1.4 Commande

Toute action effectuée par *GNU Emacs*, y compris l'insertion d'un caractère, est le résultat de l'exécution d'une commande. Un certain nombre de ces commandes sont liées à des clés :

```
forward-char              C-f
save-buffers-kill-emacs   C-x C-c
self-insert-command       A
self-insert-command       z
execute-extended-command  M-x
```

Chaque commande *GNU Emacs*, des plus simples comme les déplacements de curseur aux plus sophistiquées comme l'indentation d'un programme C, est définie par une fonction écrite en langage Lisp.

Même les caractères affichables (lettres, chiffres...) sont des clés. Ils sont tous, par défaut, liés à la même commande :

```
self-insert-command
```

Si l'on change la liaison du caractère `A` en l'associant par exemple à la commande `forward-char`, la frappe d'un `A` provoquera l'avancement du curseur au lieu de l'insertion d'un `A` dans le buffer.

En résumé :

- toute clé provoque l'exécution d'une commande;
- chaque commande provoque l'évaluation d'une fonction Lisp par un interprète Lisp interne à *GNU Emacs*.

3.2 Interaction avec *GNU Emacs*

3.2.1 Invocation d'une commande

Dans le cas d'une commande liée à une clé, il suffit de taper cette clé pour faire exécuter la commande. La définition d'un ensemble de liaisons (*bindings*) est appelé une table de clés ou *keymap*.

La *keymap*, commune à tous les buffers, est la *keymap* globale, de nom `global-map`. À chaque mode est associée une *keymap* locale, dont les définitions masquent les définitions globales. De cette façon, on peut redéfinir la valeur d'une clé en fonction du type de buffer que l'on édite.

On peut redéfinir interactivement une liaison au moyen des commandes :

```
local-set-key
local-unset-key
global-set-key
global-unset-key
```

Il est également possible d'invoquer directement une commande par son nom, au moyen de la commande `execute-extended-command`, liée à la clé `M-x`. Lors de la frappe de cette clé, le curseur est placé dans le mini-buffer, et il est ainsi possible de taper le nom de la commande à faire exécuter.

Enfin, il est possible d'invoquer l'interprète Lisp afin de lui faire interpréter une expression. Pour cela, on tape la clé `M-:` qui lance la commande `eval-expression`. Par exemple

```
M-: bf Eval : (apropos "center") RET
```

est équivalent à `M-x apropos RET center RET` et à `C-h a center RET`.

3.2.2 Paramètres de commandes

Il s'agit d'un paramètre numérique, pouvant être négatif, préfixant l'invocation d'une commande (par une clé ou par `M-x`); le paramètre est introduit au moyen de la clé `C-u`.

`C-u` [⟨*par*⟩] ⟨*clé*⟩
`C-u` [⟨*par*⟩] `M-x` ⟨*commande*⟩

Le paramètre ⟨*par*⟩ est facultatif. Lorsqu'il est absent, la commande reçoit une valeur par défaut, qui est en général le nombre 4.

```
C-u 5 C-f              (forward-char 5)
C-u C-f                (forward-char 4)
C-u 2 0 M-x goto-line  (goto-line 20)
C-u 10 C-v             (scroll-up 10)
```

3.2.3 Communication à travers le mini-buffer

- affichage de messages :

```
Auto-saving...done
Wrote /users/modele/Textes/ls.man
```

– lecture de paramètres :

`M-x goto-line` **Goto line:** `1 0 RET`
`C-x C-f` **Find file:** `/users/modele/.emacs`
`C-h v` **Describe variable:** `ctl-x-map`

Certain types de paramètres (commandes, variables, chemins...) sont associés à des mécanismes de **complétion**, provoqués par les clés `C-i`, `SPC`, et `RET`. Cela permet de ne taper que le début d'un nom, et de laisser *GNU Emacs* en compléter la fin.

3.2.4 Communication pleine page

Certains modes affichent dans le buffer une liste de choix, parmi lesquels il est possible de se déplacer, et sur lesquels il est possible d'activer des commandes. C'est par exemple le cas du mode `dired` chargé par la commande `dired` liée à `C-x d` et permettant d'éditer le contenu d'un répertoire.

Chaque ligne contient le nom d'un fichier du répertoire. Il est possible de se déplacer de ligne en ligne, de marquer les lignes avec des commandes (`d`*`elete`*, `r`*`ename`*...), d'en lancer l'exécution, de charger un autre répertoire ou un fichier (`f`*`ind`* et `v`*`isit`*), etc.

À ces modes sont associées des *keymaps* très particulières. En particulier, les caractères affichables ne sont plus liés à la commande `self-insert-command`.

3.3 Contrôle de l'environnement

3.3.1 Modes majeurs et modes mineurs

Chaque buffer est lié à un mode encore appelé **mode majeur**. Le mode majeur par défaut est le mode `fundamental-mode`. Il y a plusieurs façons de placer un buffer dans un mode particulier :

1) en chargeant dans ce buffer un fichier dont le suffixe est implicitement associé à un mode :
   - `.c .h` : modes C ou C++
   - `.scm` : mode SCHEME
   - `.el` : mode *Emacs Lisp*
2) en activant explicitement ce mode au moyen de la commande d'appel de ce mode :
   - `M-x c-mode`
   - `M-x scheme-mode`
   - `M-x emacs-lisp-mode`
   - `M-x text-mode`
   - `M-x nroff-mode`
3) en utilisant une commande associée à un mode :
   - `C-x C-b` : mode `Buffer Menu` (menu des buffers)
   - `M-x dired` : mode `dired` (édition de répertoire)
   - `M-x rmail` : mode `rmail` (lecture et envoi de courrier électronique)

Il existe également des **modes mineurs**, combinables avec un mode majeur, pour en paramétre le comportement :

- `auto-fill-mode`
- `auto-save-mode`

Chaque mode est défini par un programme *Emacs Lisp*, chargé et exécuté par *GNU Emacs*. Il est possible par conséquent possible de programmer de nouveaux modes. Les programmes *Emacs Lisp* de la distribution de base sont généralement rangés dans `/usr/local/lib/emacs/lisp`.

3.3.2 Valeur et modification des variables

Le comportement de *GNU Emacs* est paramétré par un ensemble de variables, pouvant être

- globales : communes à tous les buffers;
- locales : associées à un buffer, et donc visibles seulement dans ce buffer.

Par exemple, la variable `c-auto-newline` est globale, et utilisée par tous les buffers en mode C. Par contre, la variable `default-directory` est locale à chaque buffer et contient le chemin du répertoire courant.

Une définition locale **masque** la définition globale d'une variable de même nom. Par exemple, la variable globale `case-fold-search` peut être redéfinie localement dans un buffer.

Certaines commandes modifie implicitement la valeur d'une variable. Par exemple, la commande `compile`, qui lance une compilation dans un buffer de nom `*Compilation*`, exécute a commande contenue dans la variable `compile-command`. Il est possible de valider cette commande par défaut en tapant simplement `RETURN`, où d'entrer une nouvelle commande qui devient la valeur courante de `compile-command`.

On peut examiner le contenu d'une variable grâce à la commande `describe-variable` liée à `C-h v`.On peut obtenir la liste de toutes les variables en provoquant une complétion de nom de variable à partir du mot vide :

`M-x C-h v SPC`

La commande `set-variable` permet de modifier le contenu d'une variable. La commande `edit-options` ouvre un mode d'interaction pleine-page, le mode `options`, permettant de modifier la valeurs des variables courantes.

3.3.3 Définition de nouvelles fonctions

3.3.3.1 Éléments de *Emacs Lisp*

Il y a deux façons de définir une nouvelle fonction :

1) en nommant une macro-clavier :

`M-x name-last-kbd-macro` **Name for last kbd macro:** ⟨*nom*⟩

2) en écrivant une fonction LISP :

```
(defun ⟨nom⟩ (⟨liste de par.⟩)
    "⟨description"⟩
    (interactive ⟨mode d'interaction⟩)
    ...
)
```

Voici quelques unes des commandes *Emacs Lisp* utilisables sous *GNU Emacs* :

- `setq` : affectation d'une variable;
- `let` : déclaration et initialisation d'une variable locale;
- `if while < <= > >= =` : mise en oeuvre de tests et de boucles;
- `format` : formatage d'une chaîne de caractères;
- `insert-string` : insertion d'une chaîne de caractère dans le buffer;
- `point` : valeur du point;
- `forward-char`, `backward-char`, `beginning-of-line`, `next-line`... : toutes les commandes prédéfinies;
- `define-key` : redéfinition d'une clé (argument : une *keytab*);
- `load-library` : chargement d'une bibliothèque;

– `autoload` : spécification de chargement automatique du fichier contenant la définition d'une fonction.

Voici également quelques variables couramment utilisées :

– `global-map` : *keymap* globale à tous les modes;
– `c-mode-map c++-mode-map...` : *keymaps* locales;
– `auto-mode-alist` : liste des associations suffixe–mode
– `load-path` : chemin des répertoires contenant des fichiers de définitions *Emacs Lisp* de commandes .

3.3.3.2 Exemple

Le fichier `numeroter.el` suivant contient une définition de la fonction `numeroter` qui numérote n lignes du buffer courant.

`numeroter.el`

```
(defun numeroter (nblig)
  "Generation de numeros en debut de ligne"
  (interactive "p")
  (let (( i 1))
    (while (<= i nblig)
      (insert-string (format "%d" i))
      (setq i (+ i 1))
      (if (eobp)
          (open-line 1))
      (next-line 1)
      (beginning-of-line))))

(define-key global-map "\C-cn" 'numeroter)
```

`numeroter.el`

Il suffit de le charger par

`M-x load-file` **Load file :** ⟨*chemin*⟩

ou par

`M-x load-library` **Load library** ⟨*nom − de − fichier*⟩

pour bénéficier de cette nouvelle commande. Dans ce second cas, le fichier doit être situé dans répertoire connu de *GNU Emacs*, c'est-à-dire un des répertoires spécifiés par la variable `load-path`.

3.3.4 Les fichiers d'initialisation

3.3.4.1 Le fichier `.emacs`

Ce fichier doit être placé dans le répertoire d'accueil. Il est alors automatiquement chargé et exécuté lors du lancement de *GNU Emacs*[2].

Il est préférable de définir un fichier `.emacs` minimum et de placer l'ensemble de ses fichiers d'initialisation dans un répertoire, par exemple `~/emacs`. Dans ce cas, le fichier `.emacs` se réduit à une redéfinition de la variable globale `load-path` qui contient la liste des répertoires parcourus par *GNU Emacs* pour trouver les fichier *Emacs Lisp*, et un chargement du vrai fichier d'initialisation.

```
(setq load-path (cons (expand-file-name "~/emacs") load-path))
```

2. Il est possible de supprimer le chargement du fichier d'initialisation en lançant *GNU Emacs* avec l'option `-q`

```
(load-library ".emacs")
```

Ce programme enchaîne :

1) l'ajout de `~/emacs` en tête de la liste des répertoires de chargement;
2) le chargement de la bibliothèque `.emacs` située dans le répertoire `~/emacs`

GNU Emacs recherche successivement un fichier de nom `.emacs.elc` (version compilée par la commande `byte-compile-file`), puis `.emacs.el` (source en *Emacs Lisp*).

3.3.4.2 Fichier d'initialisation pour *GNU Emacs*

Voici un exemple de fichier d'initialisation pour *GNU Emacs*.

emacs.el

```
;;; Redefinition de cles

;;; - suspension de l'editeur (homogene avec C-x C-c)

(define-key global-map "\C-x\C-z" 'suspend-emacs)
(define-key global-map "\C-z" '(lambda ()
                                 (interactive)
                                 (message "Pour stopper emacs, taper C-x C-z")))


;;; - redéfinition de la transposition de caracteres

(define-key global-map "\C-t" 'transpose-without-move)
(load-library "transpose")


;;; Initialisation de l'environnement

(standard-display-european t)
(setq require-final-newline t)
(setq text-mode-hook '(lambda () (turn-on-auto-fill)))
(display-time)
```

emacs.el

Le fichier `transpose.el` chargé par le fichier `emacs.el` précédent contient la définition de la fonction `transpose-without-move`.

transpose.el

```
(defun transpose-without-move ()
  (interactive)
  (transpose-chars -1)
  (forward-char)
)
```

transpose.el

Exercices.

1. Exemple de chargement de fichier
   - taper `C-x`: *GNU Emacs* attend la suite;
   - taper `C-f`: *GNU Emacs* exécute la commande `find-file` en affichant dans la mini-fenêtre le message
     ```
     Find file: ~/
     ```
   - possibilité d'obtenir la liste des complétions:`?`
   - taper ensuite `/e C-i`, puis à nouveau`?`
   - taper `p C-i`: *GNU Emacs* a ouvert une seconde fenêtre, nommée `*Completions*` pour afficher la liste des completions
   - taper `a C-i`
   - taper RETURN pour valider la proposition `/etc/passwd`. La *fenêtre de complétion* est refermée et le chargement du fichier s'exécute.
2. Charger un fichier à soi et écrire dedans. Remarquer les deux `*` en bas à gauche de la fenêtre. Sauver le fichier par `C-x C-s`. Pourquoi les deux étoiles `**` ont-elles disparu? Retaper `C-x C-s`. Commenter.
3. Essayer
   ```
   C-u *
   C-u 1 0 -
   C-u C-u SPC
   C-u C-d
   C-u 1 0 DEL
   C-u C-u M-u
   ```
   Essayer avec des nombres négatifs: `C-u - 5 C-f`, `C-u - 5 C-d`, `C-u - 10 M-c`.
4. Fabriquer un fichier `ls.man` au moyen de la commande *GNU Emacs* `M-x man ls`. Éditer le fichier `ls.man`:
   - supprimer les en-têtes de pages insérées dans le fichier:
     ```
     Modified 9/22/86 ···
     LS(1) UTX/32 ···
     ```
   - mettre les titres de section avec l'initiale en majuscule
   - encadrer les titres
   - justifier à droite
   - etc,...
5. Essayer `M-x kill-line`. Utiliser le mécanisme de complétion de commandes: `M-x k C-i l C-i i C-i` (complétions également possibles avec `SPC`). Essayer `C-u 6 M-x kill-line`.
6. Taper `C-h` successivement trois fois, et commenter les différents messages. Essayer la commande d'aide `c`. Essayer ensuite `C-h c` sur plusieurs clés. Quel est le résultat pour un caractère ordinaire?
7. Essayer `C-h k` sur plusieurs clés (par exemple `M-c`, `C-x C-s`...). Essayer avec la clé `M-s`. Comment *GNU Emacs* modifie-t-il son environnement? (ouverture d'une seconde fenêtre associé à un buffer spécial appelé `*Help*`) Pour ne conserver que la fenêtre courante, taper `C-x 1`. A quelle commande est liée cette clé?
8. Faire afficher la liste des liaisons. Aller dans la seconde fenêtre au moyen de la clé `C-x o` et parcourir la liste des liaisons.
9. Changer de mode en tapant `M-x text-mode` (penser à utiliser le mécanisme de complétion). Essayer les nouvelles commandes disponibles dans ce mode. Demander de l'aide sur la clé `M-s`. Que peut-on en conclure sur le mécanisme d'aide?
10. En utilisant le mécanisme d'aide, rechercher une commande qui permet de sauver le buffer dans un nouveau fichier.
11. Justifier à droite les premières lignes de texte, en enchaînant les clés
    ```
    M-b SPC
    ```
    Puis, mémoriser cette suite de clés en tapant
    ```
    C-x ( M-b SPCC-x )
    ```
    Remarquer les messages `Defining kbd macro...` et `Keyboard macro defined`. Placer le point à la fin d'une ligne et faire exécuter la macro au moyen de la clé `C-x e`. Évaluer le nombre de blancs manquant, et invoquer la macro avec un paramètre de répétition.
    Nommer cette macro `insere-blanc`. Utiliser le mécanisme de complétion de commandes pour taper `name-last-kbd-macro`, puis pour exécuter `insere-blanc`, et remarquer le caractère dynamique de ce mécanisme dans le second cas.
12. Trouver une clé inutilisée parmi les clés `M-SPC`, `C-x SPC` et `C-c SPC` et la lier localement à la commande `insere-blanc`. Tester l'utilisation de cette nouvelle clé.

13. Définir une macro-clavier plaçant des guillemets `` et '' autour du mot sur lequel est positionné le point. Nommer cette macro `entre-guillemet` et la lier localement à une clé libre. Lier localement la macro au caractère ". Comment peut-on encore insérer le caractère "? Comment redonner au caractère " sa signification par défaut?

14. Faire afficher la page de manuel de la commande `ls` dans le buffer `*Manual Entry*` au moyen de la séquence `M-x man ls`. Tester les mécanismes de la recherche incréntale de chaînes de caractères en tapant `C-s f i l e`. Sortir de la recherche par `ESC` et rerentrer par `C-s C-s`. Itérer plusieurs recherches successives, en avant (`C-s`) et en arrière (`C-r`). Itérer les recherches jusqu'à la fin (resp. le début) du buffer et itérer une nouvelle fois. Que se passe-t-il? Quel est l'effet de `DEL` pendant la recherche.

15. En éditant le fichier `ls.man` (exercice 4, définir une macro qui remplace la chaîne `ls(1)` par `1: LS`.

16. Lire la définition des expressions régulières dans le manuel *GNU Emacs*. Utiliser la recherche sur expression régulière (`M-C-s`) pour chercher des espaces en fin de ligne.

17. Tester les mécanismes de marquage: observer l'effet des enchaînements de clés `C-SPC`, `C-v` et `C-x C-x`; essayer les positionnements sur la marque après les commandes `M-<`, `M->` et `C-s`. Itérer plusieurs fois la commande `C-u C-SPC`. Que se passe-t-il? Combien peut-on mémoriser de marques?

18. Définir une région, la détruire, et annuler la destruction par `undo`.

19. Se placer au début d'une ligne (`C-a`) et observer l'effet de

```
C-k C-y C-y C-y
revenir en début de ligne
C-k C-k C-k C-k C-y C-y
revenir en début de ligne
M-d C-y C-y
revenir en début de ligne
C-k C-k C-u 3 M-d C-n C-a C-y C-n C-n C-y
```

20. Expliquer ce qu'ont en commun les *kill commands*. Vérifier que les commandes de type *delete* lorsqu'elles sont utilisées avec un paramètre d'itération sont traitées comme des *kill* commandes. Vérifier que `C-w` est une kill commande. Que fait `M-w`?

21. Le *kill buffer* est en fait une anneau de buffers. Essayer `C-y M-y M-y...`

22. Construire le damier

```
XXXX    XXXX    XXXX    XXXX    XXXX
XXXX    XXXX    XXXX    XXXX    XXXX
    XXXX    XXXX    XXXX    XXXX    XXXX
    XXXX    XXXX    XXXX    XXXX    XXXX
XXXX    XXXX    XXXX    XXXX    XXXX
XXXX    XXXX    XXXX    XXXX    XXXX
    XXXX    XXXX    XXXX    XXXX    XXXX
    XXXX    XXXX    XXXX    XXXX    XXXX
XXXX    XXXX    XXXX    XXXX    XXXX
XXXX    XXXX    XXXX    XXXX    XXXX
    XXXX    XXXX    XXXX    XXXX    XXXX
    XXXX    XXXX    XXXX    XXXX    XXXX
XXXX    XXXX    XXXX    XXXX    XXXX
XXXX    XXXX    XXXX    XXXX    XXXX
    XXXX    XXXX    XXXX    XXXX    XXXX
    XXXX    XXXX    XXXX    XXXX    XXXX
XXXX    XXXX    XXXX    XXXX    XXXX
XXXX    XXXX    XXXX    XXXX    XXXX
    XXXX    XXXX    XXXX    XXXX    XXXX
    XXXX    XXXX    XXXX    XXXX    XXXX
```

en utilisant le moins de commandes possibles. En particulier, on n'utilisera qu'une seule fois le caractère `X` et deux fois `SPC`.

23. Observer le comportement des commandes `C-x 2`, `C-x o`, `C-x 5`, `C-x 1`, `C-x 0`. Quel est l'intérêt d'ouvrir plusieurs fenêtres sur le même buffer? Mettre en pratique en plaçant une fenêtre au début d'un buffer et une seconde à la fin du même buffer. Essayer `C-x ^` et `C-x }`.

24. Sauvegarder le contenu du buffer `*Buffer List*` dans un fichier `~/tmp/buffers`. Qu'y a-t-il de différent lorsque l'on refait afficher la liste des buffers?

25. Consulter la description du mode *Buffer Menu* et supprimer les buffers inutiles. Utiliser le menu des buffers pour sélectionner un buffer.

26. Que font les clés `C-x b` et `C-x k`?

Chapitre 4

Édition, compilation et mise au point de programmes C

4.1 Du source à l'exécutable

Sous UNIX, la plupart des programmes ont été écrits dans un langage de haut niveau, le langage C. C'est également le cas du noyau UNIX lui-même. Pour pouvoir être exécutés, ces programmes doivent d'abord être traduits en **code machine**, compréhensibles par le processeur central.

Le **fichier source**, suffixé par les caractères `.c`, est le fichier contenant le texte en C du programme. Les différentes étapes nécessaires à la construction du programme exécutable sont:

- la **compilation**: traduction du programme source en un programme équivalent, le programme **assembleur**, écrit dans un langage de bas niveau, proche de la machine;
- l'**assemblage**: construction d'un **fichier objet** qui contient la traduction du programme assembleur en code machine;
- l'**édition de liens** qui construit le programme exécutable, directement chargeable en mémoire.

Ces différentes étapes sont assurées par trois programmes: le **compilateur** C (`cc1`), l'**assembleur** (`as`), et l'**éditeur de liens** (`ld`).

En général, un programme est découpé en un ensemble de fichiers sources. La construction de l'exécutable comporte l'enchaînement des différentes étapes décrites sur la figure 4.1.

4.2 La commande gcc

Sous UNIX, la commande `gcc` permet d'enchaîner les différentes étapes de construction de l'exécutable. Voici quelques cas de figure:

- `gcc bonjour.c`: construction d'un exécutable de nom `a.out`
- `gcc bonjour.c -o bonjour`: construction d'un exécutable de nom `bonjour`
- `gcc -c bonjour.c`: construction du fichier objet `bonjour.o`
- `gcc -S bonjour.c`: construction du fichier assembleur `bonjour.s`
- `gcc main.c message.c -o message`: construction de l'exécutable `message`
- `gcc main.o message.o -o message`: édition de liens des objets et construction de l'exécutable `message`.

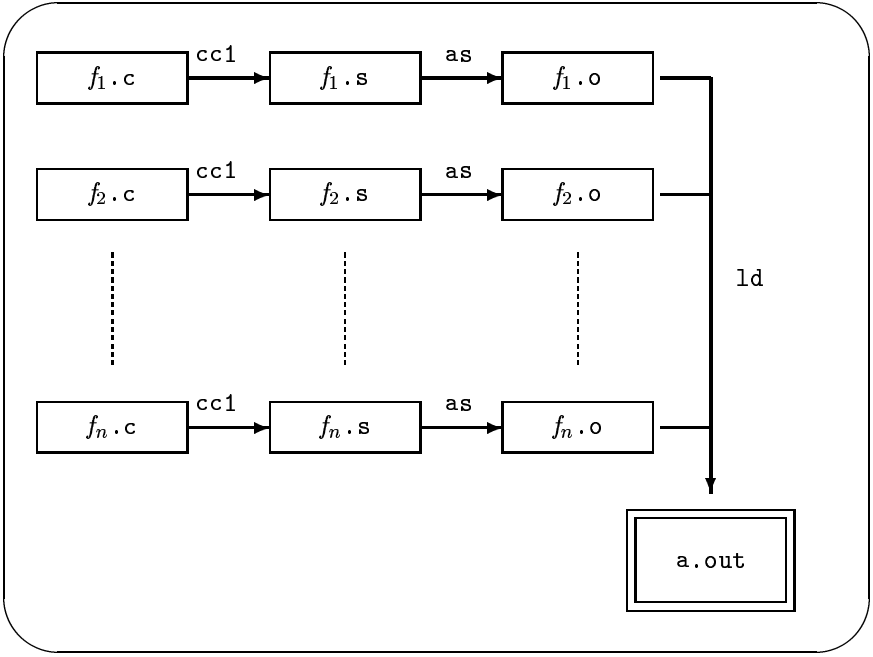


FIG. 4.1 – *Schéma de la compilation d'un programme composé de plusieurs fichiers source.*

4.3 La commande make

Il existe une façon de décrire toutes les étapes intervenant dans la construction d'un exécutable au moyen d'un fichier appelé `Makefile`.

Une commande standard, la commande `make`, permet d'enchaîner automatiquement toutes ces étapes. Pour cela, elle lit la liste des traitements à effectuer dans le fichier `Makefile`.

La commande `make` n'exécute que les étapes réellement nécessaires. Par exemple, dans le cas où seul un fichier source est modifié, elle ne recompilera pas les autres. Cela optimise la reconstruction de l'exécutable, et garantit la cohérence du résultat.

Makefile

```
CC= gcc
CFLAGS= -g
OBJ= main.o message.o

message : $(OBJ)
        $(CC) $(CFLAGS) $(OBJ) -o message

main.o : main.c
message.o : message.c message.h

clean :
        rm -f $(OBJ) message
```

Makefile

Dès qu'un programme comporte plus d'un fichier, il est **indispensable** de définir un fichier `Makefile`. Dans ce cas, les différents fichier utilisé pour ce programme (sources, objets, makefile) doivent être placés dans un répertoire qui leur est réservé.

Il est possible de générer automatiquement les dépendances au moyen de la commande `gcc` en utilisant l'option `-MM`. Par exemple, sur l'exemple précédent, l'exécution de

```
gcc -MM *.c
```

produit la liste

```
main.o : main.c
message.o : message.c message.h
```

On trouvera page 63 l'exemple d'un programme écrit en shell permettant de construire automatiquement des Makefiles simples.

4.4 Édition, compilation et mise au point de programmes C sous *GNU Emacs*

4.4.1 Édition de programmes C

Le mode C de *GNU Emacs* offre différentes facilités pour éditer un programme C. Ce mode est automatiquement activé sur tout fichier dont le nom est suffixé par `.c` ou `.h`.

Une des fonctionnalités les plus intéressantes est l'indentation automatique de programme. Une indentation correcte est fondamentale pour la lisibilité et la maintenabilité des programmes. On peut remarquer que certaines erreurs, qui ne sont pas de nature syntaxique et par conséquent sont indécelables par le compilateur, peuvent être facilement trouvées en utilisant les fonctions d'indentation.

Considérons par exemple l'instruction suivante gérant un compteur circulaire.

```
if (compteur < borne_sup)
    compteur++;
else;
    compteur = borne_inf;
```

Une erreur a été commise au moment de la frappe : le programmeur a tapé un point-virgule à la suite du mot clé `else`. De ce fait, l'instruction

```
compteur = borne_inf;
```

ne constitue plus la partie *sinon* du test, mais est une instruction indépendante toujours exécutée à la suite du test. Une indentation automatique sous *GNU Emacs* transforme le code de l'exemple en :

```
if (compteur < borne_sup)
    compteur++;
else;
compteur = borne_inf;
```

texte pour lequel l'erreur est immédiatement décelable.

L'indentation d'une ligne se fait au moyen de la commande `c-indent-command` liée à la clé `C-i` ou `TAB`. L'indentation d'une région se fait au moyen de la commande `indent-region` liée à la clé `M-C-\`. Le programme suivant ne respecte aucune règle d'indentation. Il est illisible.

mal-indente.c

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

static void usage(char *s);

   int main(int argc, char *argv[])
{
```

```
 double a;
   double b;
double c;
 double delta;

 if (argc != 4)
{usage(argv[0]);
exit(EXIT_FAILURE);}

   a = atof(argv[1]);
b = atof(argv[2]);
c = atof(argv[3]);
    delta = b*b - 4*a*c;

    if (delta > 0)
        {
        double racine_delta = sqrt(delta);

        printf("Deux racines: %g et %g\n",
        (-b - racine_delta)/(2*a),
        (-b + racine_delta)/(2*a));
           }
           else
           {
           if (delta == 0)
           printf("Une racine: %g\n", -b/(2*a));
        else
/* delta < 0 */
            printf("Pas de racine reelle.\n");
          }
    return EXIT_SUCCESS;
        }

        static void usage(char *s)
        {
        fprintf(stderr, "Usage: %s a b c\n", s);
}
```

mal-indente.c

L'indentation par défaut réalisée par *GNU Emacs* produit le résultat suivant :

bien-indente.c

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

static void usage(char *s);

int
main(int argc, char *argv[])
{
  double a;
  double b;
  double c;
  double delta;

  if (argc != 4)
    {
      usage(argv[0]);
      exit(EXIT_FAILURE);
    }
```

```
  a = atof(argv[1]);
  b = atof(argv[2]);
  c = atof(argv[3]);
  delta = b*b - 4*a*c;

  if (delta > 0)
    {
      double racine_delta = sqrt(delta);

      printf("Deux racines: %g et %g\n",
             (-b - racine_delta)/(2*a),
             (-b + racine_delta)/(2*a));
    }
  else
    {
      if (delta == 0)
        printf("Une racine: %g\n", -b/(2*a));
      else
        /* delta < 0 */
        printf("Pas de racine reelle.\n");
    }
  return EXIT_SUCCESS;
}

static void
usage(char *s)
{
  fprintf(stderr, "Usage: %s a b c\n", s);
}
```

bien-indente.c

4.4.2 Lancement d'une compilation

Il est possible de lancer la compilation d'un programme C directement depuis *GNU Emacs* au moyen de la commande `compile`. Par défaut cette commande n'est liée à aucune clé, aussi doit-on taper

```
M-x compile
```

GNU Emacs ouvre une fenêtre de compilation sur un buffer de nom `*Compilation*` dans lequel il lance par défaut la commande `make`. Il est bien sûr possible de redéfinir le traitement associé à `compile`. S'il y a des erreurs de compilation, les messages associés sont insérés dans le buffer.

4.4.3 Recherche d'erreurs

Il est particulièrement intéressant de compiler ses programmes sous *GNU Emacs*. En effet, la commande `next-error`, liée à la clé `C-x‘`, permet de se positionner automatiquement sur les éventuelles erreurs. On parcourt les différentes erreurs en itérant l'utilisation de cette clé. Pour chaque erreur de syntaxe rencontrée, *GNU Emacs* positionne le curseur sur la ligne du programme correspondante. Cela permet des corrections très rapides des erreurs de syntaxe.

Lorsque toutes les erreurs de syntaxe sont corrigées, il reste souvent des erreurs de programmation. On utilisera pour terminer la mise au point le débogueur symbolique `gdb` directement depuis *GNU Emacs*. On se réfèrera à la documentation GNU de gdb.

Exercices.

1. Charger un fichier C sous *GNU Emacs* et tester les mécanismes d'indentation (`C-i`, `M-C-\`, ...). En particulier, tester l'effet de la suppression sur une ligne précédente d'un point-virgule, d'un guillemet, d'une quote, d'une parenthèse, ...
2. Compiler un programme C sous *GNU Emacs*, au moyen de la commande `M-x compile` (on se placera dans un répertoire contenant un fichier `Makefile` et plusieurs fichiers sources). Rechercher et corriger les erreurs de syntaxe au moyen de la clé `C-x ‘` (si le programme ne contient pas d'erreurs, en insérer et tester les recherches).

Chapitre 5

Programmation en shell

Sous UNIX, le terme de **shell** désigne une famille d'interprètes de commandes, dont les principaux sont :

- sh $\longrightarrow \left\{ \begin{array}{l} \text{ksh} \\ \text{bash} \end{array} \right.$
- csh $\longrightarrow$ tcsh

Un *shell* est une commande UNIX ordinaire, pouvant être lancée depuis l'interprète courant. Un interprète peut ainsi lancer une nouvelle occurrence de lui-même. On parle dans ce cas d'**empilement de shell**.

5.1 Fonctionnement général

Le fonctionnement général d'un *shell* est le suivant :

1: phase de saisie. Cette phase concerne la lecture d'une **ligne de commande**; elle peut mettre en jeu différents mécanismes d'aide à la saisie : édition de la ligne, historique, complétions...

2: phase de substitution. La *ligne de commande* est analysée et différents types de substitutions sont effectués; par exemple, le motif `*.c` est remplacé par la liste de tous les noms de fichiers du répertoire courant terminés par les caractères `.c` (voir 2.3.4).

3: phase d'exécution. La **commande** résultant de la phase de substitution est recherchée puis exécutée.

De plus, les *shells* mettent en oeuvre des fonctionnalités diverses comme :

- la gestion d'un **environnement** permettant de paramétrer l'exécution des commandes et le comportement du *shell* lui même;
- le traitement **commandes internes** (structures de contrôle, interaction avec le système...
- des mécanismes de **redirections**;
- l'exécution de fichiers de commandes appelés **shell scripts**, ou **scripts**.

5.2 Structure et traitement d'une commande

5.2.1 Décomposition d'une commande

Une **commande** est une *ligne de commande* dans laquelle les substitutions ont été résolues. Par exemple, la ligne de commande

```
date >RC.L; ls -l .??* | grep rc >>RC.L
```

pourra être transformé, après substitution, en la commande

```
date >RC.L; ls -l .bash_history .bash_profile .bashrc .emacs .mailrc | grep rc >>RC.L
```

Une commande peut être décomposée en une liste de *pipelines*, chacun étant lui même formé d'un enchaînement de *commandes simples*. La commande de l'exemple précédent se décompose en une liste de deux pipelines :

`date >RC.L` ; `ls -l .bash_history .bash_profile .bashrc .emacs .mailrc | grep rc >>RC.L`

Le premier est réduit à une commande simple; le second est composé de deux commandes simples :

`ls -l .bash_history .bash_profile .bashrc .emacs .mailrc` | `grep rc >>RC.L`

5.2.2 Commande simple

5.2.2.1 Structure d'une commande simple

La forme générale d'une **commande simple** est la suivante :

$$\underbrace{\langle mot_0 \rangle}_{\text{nom de la commande}} \quad \underbrace{\langle mot_1 \rangle \ \langle mot_2 \rangle ...}_{\text{arguments}} \ [\langle redirections \rangle]$$

5.2.2.2 Exécution d'une commande simple

Lors de la phase d'exécution, chaque commande simple est découpée en une liste de mots, délimités par des **séparateurs de champs** qui sont par défaut les caractères `SPC`, `TAB`, `NEWLINE`. La recherche de la commande associée au premier mot $\langle mot_0 \rangle$ s'effectue de la façon suivante :

Si $\langle mot_0 \rangle$ est une *commande interne*
elle est directement exécutée par le *shell*

sinon un fichier de nom $\langle mot_0 \rangle$ est recherché dans une liste de répertoires de fichiers exécutables. Plusieurs cas sont possibles :

- le fichier n'est pas trouvé :

```
$ hello
hello: command not found
```

- le fichier existe, mais l'utilisateur n'y a pas accès en exécution :

```
$ main.c
main.c: Permission denied
```

- le fichier est un programme exécutable avec accès en exécution (par exemple `cp`, `rm`...) : lancement de la commande;
- le fichier est un script avec accès en exécution : lancement d'un *shell* qui interprète ce script; le chemin absolu du *shell* lancé peut être spécifié par la première ligne du script :

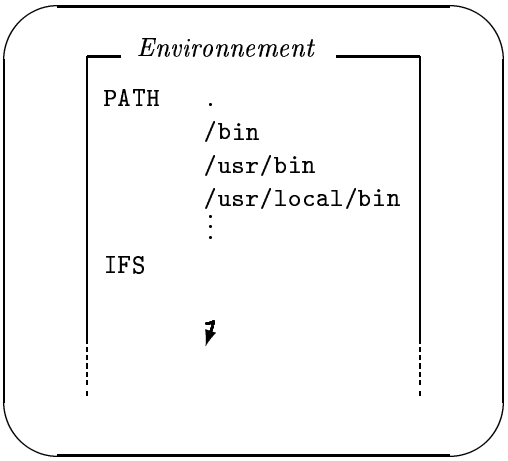
```
#!/bin/sh
#!/bin/csh
...
```

5.2.2.3 Environnement

L'**environnement** (voir également 5.4) est composé d'une liste de couples

$$\prec variable, chaîne \succ$$

Certaines variables de l'environnement paramètrent le comportement du shell. Par exemple, les variables `PATH` et `IFS` définissent respectivement la liste des chemins de répertoires parcourus par le *shell* pour rechercher le fichier correspondant à un nom commande, et la liste des séparateurs de mots.



5.2.2.4 Traitement des redirections

Les redirections permettent de modifier la définition par défaut des entrées et sorties standard (voir 2.2.3) de la commande lancée (script ou exécutable). Les redirections les plus utilisées sont :

`<`	:	redirection de l'entrée standard dans un fichier
`>`	:	redirection de la sortie standard dans un fichier
`>>`	:	idem, avec positionnement à la fin du fichier s'il existe
`<<`⟨*marqueur*⟩	:	redirection de l'entrée standard dans un script
`2>`	:	redirection de la sortie erreur dans un fichier
`2>&1`	:	fusion de la sortie erreur sur la sortie standard

```
$ cat redir
#!/bin/sh

wc << EOF
ls -l
pwd
xxxx
EOF
yyy
$
$ redir
       3       4      15
./redir: yyy: not found
$
```

Sur cet exemple, toutes les lignes entre la redirection (`<<EOF`) et le marqueur de fin de redirection (`EOF` en début de ligne) ont été envoyées sur l'entrées standard de `wc`, qui compte le nombre de lignes,

mots et caractères lus sur son entrée standard. Les lignes situées après le marqueur sont exécutées normalement.

5.2.2.5 Valeur d'une commande

Toute commande qui termine son exécution retourne une **valeur**. Il y a deux sortes de terminaisons :

1) *terminaison normale* : le processus exécute un appel système `exit(`***i***`)` ; c'est ce qui se passe par défaut après l'exécution de la dernière instruction d'un programme;
2) *terminaison anormale* : le processus est interrompu par la réception du signal numéro ***s***, par exemple le signal 9 sur un `kill -9` (voir 2.4.5).

La valeur d'une commande est conditionné par sa terminaison :

1) dans le cas d'une terminaison normale, c'est la valeur ***i*** du paramètre de l'appel `exit` (normalement 0);
2) dans le cas d'une terminaison anormale, la valeur est un mot | ***c*** | ***s*** |, où ***s*** est le numéro du signal ayant provoqué la terminaison et ***c*** est un entier valant 1 si un fichier `core` a été produit et 0 sinon.

La valeur d'une commande interprété par le *shell* comme un booléen. La valeur par défaut, *zéro*, correspond à *vrai*. Les autres correspondent à faux.

```
$ cat exitwith.c
main(int argc, char *argv[])
{
    if (argc == 2)
        exit(atoi(argv[1]));
}
$ if exitwith 0
> then echo oui
> else echo non
> fi
oui
$ if exitwith 1
> then echo oui
> else echo non
> fi
non
$
```

La valeur peut être récupéré sous le *shell* au moyen du motif `$?` qui contient à tout instant la valeur de la dernière commande exécutée :

```
$ exitwith 100
$ echo $?
100
$ echo $?
0
$
```

Remarque 1 *On appellera le* **résultat** *d'une commande tout ce quelle affiche sur sa sortie standard. Par exemple, la valeur de*

`pwd`

est 0, tandis que son résultat est une chaîne de la forme

```
/users/modele/tmp
```

5.2.3 Pipeline

5.2.3.1 Structure d'un pipeline

Un **pipeline** est un enchaînement de commandes simples séparées par des `|`.

```
ps -ax | grep bash | grep -v bash | more
```

5.2.3.2 Exécution d'un pipeline

Toutes les commandes simples du pipeline sont exécutées en parallèle.

5.2.3.3 Traitement des redirections

La sortie standard d'une commande située à gauche du symbole `|` est redirigée dans l'entrée standard de celle située à sa droite (voir 2.2.3).

5.2.3.4 Valeur d'un pipeline

La valeur d'un pipeline est celle de la commande la plus à droite.

5.2.4 Liste de pipelines

5.2.4.1 Structure d'une liste

On appelle **liste de pipelines**, ou **liste de commandes**, ou encore **liste** un enchaînement de pipelines ou de commandes simples séparés par les opérateurs de listes :

`;`	:	enchaînement séquentiel
`&`	:	enchaînement parallèle
`&&`	:	et
`||`	:	ou

Les deux premiers opérateurs peuvent être suivis d'une commande vide :

```
cp * .. &
cd $CHEMIN && ls
a.out 2>&1 >LOG || echo "Erreur"
```

5.2.4.2 Exécution d'une liste

$\langle cmd_1 \rangle$ `;` $\langle cmd_2 \rangle$	:	$\langle cmd_2 \rangle$ est lancée lorsque $\langle cmd_1 \rangle$ est terminée;
$\langle cmd_1 \rangle$ `&` $\langle cmd_2 \rangle$	:	$\langle cmd_2 \rangle$ et $\langle cmd_1 \rangle$ sont lancées en parallèle;
$\langle cmd_1 \rangle$ `&&` $\langle cmd_2 \rangle$	:	$\langle cmd_2 \rangle$ est lancée si $\langle cmd_1 \rangle$ retourne *vrai*;
$\langle cmd_1 \rangle$ `||` $\langle cmd_2 \rangle$	:	$\langle cmd_2 \rangle$ est lancée si $\langle cmd_1 \rangle$ retourne *faux*;

5.2.4.3 Traitement des redirections

Le traitement des redirection est intégralement effectué au niveau de la commande simple ou du pipeline. Il n'y a pas de redirections entre deux pipelines.

5.2.4.4 Valeur d'une liste

La valeur d'une liste est celle du *pipeline* le plus à droite.

5.2.5 Parenthésage

Le parenthésage au moyen des parenthèses provoque l'exécution de la commande placée entre les parenthèses dans un sous-shell. Cela permet par exemple de rediriger les entrées-sorties d'une liste de commandes :

```
( cat $F1 || cat $F2 ) | wc -l

( date ; pwd ; ls -CF ) >LOG
```

5.3 Mécanismes de substitution

5.3.1 La phase de substitution

La phase de substitution précède la phase d'exécution. Il y a plusieurs sortes de substitution, chacune étant associée à un ou plusieurs **motifs**. Par exemple, on a vu en 2.3.4 qu'un mot contenant une ou plusieurs occurrences du caractère * est un motif de *substitution de chemin*.

L'interprète recherche les motifs à remplacer dans la *ligne de commande* et remplace chacun d'entre eux par le résultat de la substitution. La phase de substitution est elle même découpée en trois sous-phases :

- remplacement de commandes
- remplacement de variables
- remplacement de chemins

effectuées dans cet ordre. Le remplacement d'une commande peut mettre en oeuvre récursivement une nouvelle phase de substitution.

5.3.2 Remplacement de commandes

Le motif d'un remplacement de commandes est

$$`\langle cmd \rangle`$$

Il est remplacé par le *résultat* de la commande $\langle cmd \rangle$.

```
$ echo `date`: `who | wc -l` util. sur `hostname`
Fri Jan 17 11:14:51 EET 1992: 7 util. sur glenn
$
```

5.3.3 Remplacement de variables et paramètres

Les variables *shell* sont toutes des chaînes de caractères. Un identificateur est formé d'une lettre suivie de lettres ou de chiffres. Le caractère _ est considéré comme une lettre. Certaines variables sont prédéfinies, comme la variable `PATH` ou la variable `IFS` (voir 5.2.2 ou 5.4).

L'affectation d'une variable se fait au moyen de la construction

$$\langle ident \rangle = \langle mot \rangle$$

Le caractère `$` permet de remplacer la variable par sa valeur :

```
$ RAC=/bin
$ echo $RAC
/bin
$ echo $RAC/pwd
/bin/pwd
$ $RAC/pwd
/users/modele/emacs
$
```

Il existe également un vecteur de variables que l'on peut affecter au moyen de la commande `set` :

`set` $\langle mot_1 \rangle$ $\langle mot_2 \rangle$ `...`

place respectivement $\langle mot_1 \rangle$ dans la première case du vecteur, $\langle mot_2 \rangle$ dans la seconde, etc.

Seuls les neuf premiers éléments sont directement accessibles. Les motifs associés à la manipulation du vecteur sont les suivants :

`$#`	:	nombre d'éléments courant
`$1`	:	valeur du premier élément
	⋮	
`$9`	:	valeur du neuvième élément
`$*`	:	liste de tous les éléments
`$@`	:	équivalent à `"$*"`

La commande `shift` permet d'accéder, lorsqu'ils existent, aux autres éléments (voir 5.7.2).

```
$ set ‘date‘
$ echo $*
Fri Jan 17 14:45:41 EET 1992
$ echo $#
6
$ echo Il est $4
Il est 14:45:41
$
```

Lors de l'exécution d'un script, le vecteur de variables est initialisé avec la liste des paramètres éventuels :

```
$ cat arg
echo
echo $# arg: $*
echo Premier: $1
echo
$
$ arg

0 arg:
Premier:

$ arg a b c

3 arg: a b c
Premier: a

$
```

Il existe d'autres motifs associés au caractère `$`:

`$0`	:	nom de la commande courante (shell courant ou script)
`$?`	:	valeur de retour de la dernière commande exécutée (voir page 46)
`$$`	:	*pid* de la commande courante
`$!`	:	*pid* du dernier processus détaché

Enfin, les accolades peuvent être utilisées pour résoudre certaines ambiguïtés:

```
$ PREFIXE="TMP_"
$
$ N1=${PREFIXE}1
$ N2=${PREFIXE}2
$
$ echo $N1 $N2
TMP_1 TMP_2
$
```

5.3.4 Remplacement de chemins

Il existe trois motifs de remplacement de chemins:

`*`	:	suite éventuellement vide de caractères
`?`	:	un caractère quelconque
`[]`	:	alphabet (ensemble de caractères)
		`[abcdef]` : $\{a,b,c,d,e,f\}$
		`[a-f]` : //
		`[A-Za-z]` : toutes les lettres

Le caractère `.` en tête d'un nom de fichier subit un traitement particulier. Un tel nom de fichier n'est pas reconnu par un motif commençant par `*` ou `?`. En général, ces fichiers sont des fichiers de configuration ou de données utilisés par des logiciels, qui ne seront accédés que s'ils sont spécifiés explicitement. Par conséquent, le motif `*` produit la liste de tous les fichiers du répertoire courant ne commençant pas par un point, et `.*` celle de tous les fichiers commençant par un point (y compris les répertoires `.` et `..`).

```
$ echo /bin/[a-c]?
/bin/ar /bin/as /bin/cc /bin/cp
$ (cd /bin; echo *z* )
pagesize size
$
```

5.3.5 Quotation

Les mécanismes de **quotation** permettent de bloquer la remplacement de certains motifs. Le caractère `\` transforme le caractère qui le suit en caractère normal. Une suite de caractères entre *quotes* `'` n'est pas interprétée, et est considéré comme un seul mot.

```
$ echo \*
*
$ echo '/***\'
```

```
/***\
$
$ echo '/bin/[a-c]?'
/bin/[a-c]?
$ N='/bin/[a-c]?'
$ echo $N
/bin/ar /bin/as /bin/cc /bin/cp
$
```

Une chaîne placée entre guillemets est partiellement quotée; seuls les motifs de remplacement de chemin sont inhibés.

```
$ echo "*$0*"
*-bash*
$
$ N='/bin/[a-c]?'
$ echo $N
/bin/ar /bin/as /bin/cc /bin/cp
$ echo "$N"
/bin/[a-c]?
$
```

5.4 Environnement

Il s'agit d'un ensemble de variables pouvant être utilisées

1) pour paramétrer le comportement du *shell* (`PATH`, `HOME`, `PS1`, `PS2`, `IFS`, `MAIL`...); ces variables sont en général prédéfinies;
2) pour mémoriser des chaînes et écrire des programmes;
3) pour paramétrer le comportement des commandes lancées (`TERM`, `EDITOR`...).

Par défaut, les variables sont locales au *shell*. Pour être accessibles depuis une autre commande (le troisième cas), elles doivent être explicitement déclarées globales au moyen d'un mécanisme d'**exportation**. Une fois exportées, elles sont recopiées dans l'environnement de toute commande ou de tout *sous-shell* lancé. Un effet de bord dans l'environnement d'un *shell* fils n'affecte pas celui de son père.

```
$ A=un
$ sh
$ echo /$A/
//
$ ^D
$
$ export A
$ sh
$ echo /$A/
/un/
$ A=deux; B=trois
$ echo /$A/$B/
/deux/trois/
$ ^D
$
$ echo /$A/$B/
/un//
```

```
$
```

Il est également possible de définir un *environnement temporaire* dans lequel sera exécutée une commande. Par exemple, la commande

```
CFLAGS="-O -g" make
```

construit un environnement temporaire composé des variables exportées et de la variable `CFLAGS` pour faire exécuter la commande `make`.

L'environnement peut être initialisé au moyen de fichiers *start-up* lus par le shell au moment de la connexion. Ces fichiers diffèrent d'un *shell* à l'autre :

```
sh    :  .profile
csh   :  .login
         .cshrc
bash  :  .bash_profile
         .bashrc
```

```
HOSTNAME=‘hostname‘
PS1="$HOSTNAME: "
PS2="$HOSTNAME? "
export PS1 PS2
```

5.5 Commandes UNIX pour la programmation en shell

5.5.1 La commande test

La commande `test`, s'utilise de deux façons différentes [1] :

`test` ⟨arg_1⟩ ⟨arg_2⟩ `...`
`[` ⟨arg_1⟩ ⟨arg_2⟩ `...` `]`

Elle permet de tester l'existence et la nature de fichiers :

`test` { `-f` | `-r` | `-w` | `-s` | `-d` } ⟨*chemin*⟩

permet de tester si ⟨*chemin*⟩ est respectivement

un fichier existant
un fichier accessible en lecture
un fichier accessible en écriture
un fichier de taille non nulle
un répertoire

```
if [ -d $NOM ]
then
    echo $NOM est un repertoire
else
    rm -i $NOM
```

1. Les deux chemins `/bin/[` et `/bin/test` sont généralement des liens sur la même commande.

```
fi
```

Elle permet également de comparer des chaînes quelconques ou des chaînes numériques :

– test d'égalité de deux chaînes :

$$\texttt{test}\ \langle chn_1 \rangle \begin{cases} = \\ != \end{cases} \langle chn_2 \rangle$$

– comparaison deux chaînes numériques :

$$\texttt{test}\ \langle chn_1 \rangle \begin{cases} \texttt{-eq} \\ \texttt{-ne} \\ \texttt{-gt} \\ \texttt{-ge} \\ \texttt{-lt} \\ \texttt{-le} \end{cases} \langle chn_2 \rangle$$

```
if [ $# -ge 1 ]
then
    echo "Usage: $0 fichier fichier
    exit 1
fi
```

5.5.2 La commande expr

La commande **expr** permet d'évaluer des expressions arithmétiques et logiques simples (sans parenthésage). Elle reçoit en argument une expression arithmétique ou logique et produit le résultat de son évaluation sur sa sortie standard.

Voici quelques uns des opérateurs qu'elle reconnaît.

$$\texttt{expr}\ \langle e_1 \rangle \begin{cases} \texttt{+} \\ \texttt{-} \\ \texttt{*} \\ \texttt{/} \\ \texttt{\%} \\ \texttt{<} \\ \texttt{<=} \\ \ldots \end{cases} \langle e_2 \rangle$$

On peut par exemple incrémenter la valeur d'une variable en écrivant :

```
NB=‘expr $NB + 1‘
```

5.5.3 La commande tr

La commande **tr** est un filtre permettant de remplacer des caractères dans un flot de données. Elle reçoit en argument deux chaînes de caractères. La première chaîne indique la liste des caractères à remplacer. La seconde chaîne donne, pour chaque caractère de la première chaîne le caractère par lequel il doit être remplacé. Si la seconde chaîne est trop courte, elle est automatiquement rallongé en répétant son dernier caractère. La notation `[`x`-`y`]` permet de spécifier la suite de caractères allant de x à y dans la table ASCII. Un caractère non affichable peut être désigné par son code ASCII exprimé en base 8.

Voici trois exemples d'exécution de la commande `tr`. Le symbole C-d indique la frappe d'un caractère `C-d` fermant le flot d'entrée et provoquant la terminaison du processus. Pour plus de clarté, les lignes affichées par `tr` sont typographiés en italique.

```
$ tr Aa aA
abc Abc AaA
Abc abc aAa
C-d
$
$ tr [A-Z] [a-z]
La commande TR(1)
la commande tr(1)
remplace des CARACTERES
remplace des caracteres
C-d
$
$ tr ' \011' -
hhh     h  iii
hhh-h--iii
$
```

L'option `-c` indique à `tr` de prendre le complément de la première chaîne comme alphabet d'entrée. L'option `-d` demande la suppression dans le flot de toute occurrence d'un caractère de la première chaîne. Enfin, l'option `-s` entraine le remplacement dans le flot de chaque occurrence $cc\cdots c$ d'un caractère de la seconde chaîne par le seul caractère c. Dans l'exemple suivant, toutes les suites de caractères différentes d'une lettre, d'un chiffre ou d'un caractère souligné par un caractère `NEWLINE`.

```
$ tr -cs [A-Za-z0-9_] '\012'
main (int argc, char **argv) { exit(1); }
main
int
argc
char
argv
exit
1
C-d
$
```

5.5.4 La commande sed

La commande `sed` permet d'effectuer des traitements de texte simple sur chaque ligne du flot. Nous ne décrivons pas ici le fonctionnement complet de cette commande. Nous présentons seulement une de ses utilisations les plus fréquentes : le remplacement de motif. Un motif est construit au moyen d'opérateurs, par exemple :

- `*` qui indique la répétition du caractère précédent un nombre quelconque de fois,
- `.` qui désigne un caractère quelconque sauf `NEWLINE`,
- `^` qui, placé en début de motif, signifie «*début de ligne*»,
- `$` qui, placé en fin de motif, signifie «*fin de ligne*»,
- `[`$x_1x_2\ldots x_n$`]` qui désigne l'alphabet $\{x_1,x_2,\ldots,x_n\}$,
- `[`x_1`-`x_2`]` qui désigne l'alphabet composé des codes ascii allant de x_1 à x_2,

– `[^···]` qui désigne le complément de l'alphabet décrit par `[···]`,

– `\` qui permet d'utiliser un opérateur comme un caractère ordinaire.

Un tel motif est appelé une **expression régulière**. Les expressions régulières unix sont décrite dans la page de manuel de la commande `ed`. Voici quelques exemples d'expressions régulières UNIX :

– `^[^a-zA-Z]*` : suite située en début de ligne de caractères non-alphabétiques;

– `//.*` : suite de caractères commençant par `//` et allant jusqu'à la fin de la ligne;

– `:[^:]*:` : suite maximale de caractères commençant et finissant par `:` et ne contenant ni `:` ni `NEWLINE`.

Le remplacement de motif se met en oeuvre de la manière suivante :

`sed -e s/`⟨*motif*⟩`/`⟨*chaîne*⟩`/`⟨*option*⟩

Par défaut, la substitution n'est effectuée qu'une fois par ligne. L'option `g` (global) demande d'effectuer toutes les substitutions possibles. Le caractère `/` sert de délimiteur. Il peut être remplacé par n'importe quel autre caractère n'apparaissant pas dans le motif ni dans la chaîne de remplacement. Le caractère `&` dans la chaîne de remplacement représente la chaîne de caractères remplacée.

Par exemple, la commande

```
sed -e 's/^[^a-zA-Z]*//'
```

recevant le flot d'entrée

```
un
  deux
...trois...
```

le recopie en

```
un
deux
trois...
```

Il est possible d'enchaîner plusieurs substitutions. Voici un exemple traduisant des commentaires C++ en commentaires C. On notera l'utilisation du caractère `:` à la place du caractère `/` comme délimiteur.

```
sed -e 's://.*:/*& */:' -e 's://::'
```

Cette commande recevant le flot

```
// Fichier de
// demonstration

main() {}
```

produit le résultat

```
/* Fichier de */
/* demonstration */

main() {}
```

Cet exemple doit être considéré comme un cas d'école car le mécanisme de traduction pourrait introduire des erreurs dans un programme de grande taille, par exemple un programme contenant la

suite `//` au sein d'une chaîne de caractères. Pour des traductions de ce niveau de complexité, il est préférable d'utiliser des outils comme `lex`, `flex`, `yacc` ou `bison`.

5.6 Fonctions

Un programme en shell peut être décomposé en fonctions. Une fonction est définie en utilisant la syntaxe du langage C. Voici un exemple de gestion de messages et d'erreurs. L'utilisation des fonctions est indispensable pour modulariser le programme et éviter les duplications de code.

```
display_usage ()
{
       echo Usage: $0 [-acfx] fichier...
}

display_help ()
{
       display_usage
       cat << '__EOF__'
  -v : visualisation des calculs
  -r : traitement récursif
__EOF__
}

usage ()
{
       display_usage
       exit 1
}
```

Le mot clé `buitin` précédant un nom de commande $\langle nc \rangle$ permet de faire référence à la commande originale $\langle nc \rangle$, même si celle-ci a été redéfinie par une fonction. On peut transmettre des paramètres à une fonction. Ils sont récupérables au moyen du mécanisme de remplacement de paramètres (voir page 49).

```
cd ()
{
        builtin cd $1
        pwd
}
```

5.7 Structures de contrôle

Les *shells* disposent, parmi leurs commandes internes, de structures de contrôle classiques : boucles, tests, aiguillages et échappements. Nous allons passer rapidement en revue les structures de contrôle du *shell* standard `sh`, qui sont aussi celles des *shells* `bash` et `ksh`.

5.7.1 Les tests

Le test *si-alors-sinon* s'écrit :

```
if ⟨liste⟩
then
    ⟨liste⟩
[ elif ⟨liste⟩
then
    ⟨liste⟩
        ⋮ ]
[ else
    ⟨liste⟩ ]
fi
```

```
if [ $USER = "$1" ]
then
    echo cést moi
else
    ce nést pas moi
fi
```

L'aiguillage permet d'enchaîner des tests en comparant un mot avec une liste de motifs :

```
case ⟨mot⟩ in
  ⟨motif⟩ [ | ⟨motif⟩ ... ])
    ⟨liste⟩
      ...
    ;;
    ...
esac
```

```
case $N in
  mbox )
    echo $N: boite aux lettres
   ;;
  *~ | .*~ )
#    echo $N: emacs backup
   ;;
    ⋮
  * )
    echo $N: unknown type
   ;;
esac
```

5.7.2 Itérations

La boucle `while`

```
while ⟨liste1⟩
do
    ⟨liste2⟩
    ...
done
```

itère l'exécution de ⟨$liste_2$⟩...tant que ⟨$liste_1$⟩ retourne la valeur vrai.

L'exemple suivant utilise la commande interne `shift` qui décale le vecteur des paramètres d'une position vers la gauche, le premier élément étant perdu. La boucle est répétée tant que le nombre de paramètres est positif.

```
while [ $# -gt 0 ]
do
    case $1 in
      -o )
        shift
        $OUTPUT=$1
       ;;
    esac
    shift
done
```

La boucle `for` permet de répéter un traitement pour tous les éléments d'une liste.

```
for ⟨nom⟩ [ in ⟨mot1⟩ ... ]
do
    ⟨liste⟩
        ⋮
done
```

Par exemple, la commande suivante

```
for N in *
do
    cp $N $N.svg
done
```

répète la commande

`cp` ⟨*nom*⟩ ⟨*nom*⟩`.svg`

pour ⟨*nom*⟩ prenant successivement pour valeur tous les noms de fichier du répertoire courant.

La construction

`for N in ‘cat $LISTE‘; do ...; done`

permet d'itérer un traitement sur tous les mots de la liste de mots contenue dans le fichier `$LISTE`.

5.7.3 Échappements

Il existe trois échappements :

`continue`	:	passage à l'itération suivante
`break` [⟨n⟩]	:	sortie de ⟨n⟩ itération(s)
`exit` [⟨n⟩]	:	sortie du programme avec la valeur ⟨n⟩

5.8 Commandes internes

Diverses commandes sont traitées directement par le *shell*; ce sont les commandes **internes**. Nous avons déjà rencontré `cd`, `echo`, `shift`, `set`, etc. Citons également les commandes suivantes :

`:`	:	retourne *vrai* (utilisé pour les boucles infinies)
`.`	:	exécution locale
`source`	:	//
`read`	:	lecture sur l'entrée standard
`eval`	:	évaluation d'une chaîne
`exec`	:	remplacement du shell courant

L'exemple suivant montre comment écrire, en *shell*, un *shell* dont les possibilités sont volontairement restreintes. Ce programme s'appelle `sh--`. Il effectue une boucle infinie dans laquelle il itère :

- lecture d'une commande;
- rejet de la commande si elle tente de changer de répertoire ou d'exécuter une commande dans un répertoire autre que ceux spécifiés dans la variable `PATH`;
- évaluation de la commande si elle est correcte.

sh--

```
#!/bin/sh

echo "*Restricted shell: you can only execute local or standard commands*"

while :
do
    echo -n $PS1
    read COMMAND ARGS

    case $COMMAND in

      PATH=* )
        echo "Sorry, you cannot redefine your path"
        ;;

      cd* )
        echo "Sorry, you cannot leave this directory"
        ;;

      */* )
        echo "Sorry, you can only execute standard or local commands"
        ;;

      sh | csh | bash )
        echo "Sorry, you cannot use another shell"
        ;;

      "" )
        echo "Use \"exit\" to leave sh--"
        ;;

      exit | bye )
        break 1
        ;;

      * )
        eval $COMMAND $ARGS
        ;;

    esac
```

```
done

echo Bye.
```

sh--

Voici un exemple d'exécution de cette commande.

```
$ sh--
*Restricted shell: you can only execute local or standard commands*
$ cd
Sorry, you cannot leave this directory
$ ../../sh
Sorry, you can only execute standard or local commands
$ bash
Sorry, you cannot use another shell
$ PATH=../..
Sorry, you cannot redefine your path
$ ls /bin/a*
/bin/adb
/bin/ar
/bin/as
$ A="un test"
$ echo $A
un test
$
Use "exit" to leave sh--
$ exit
Bye.
$
```

Ce shell restreint peut facilement être trompé et il est par exemple possible d'effectuer une manipulation, autre qu'une interruption clavier, permettant de lancer un vrai shell. Comment ?

Remarque 2 *Si l'on remplace la ligne*

```
eval $COMMAND $ARGS
```

par

```
$COMMAND $ARGS
```

les remplacement de variables et de commandes ne seront plus effectuées. En effet, considérons le cas de l'évaluation de la ligne

```
echo $A
```

de l'exemple précédent. Dans ce cas

```
$COMMAND $ARGS
```

est réécrit en

```
echo $A
```

puis exécuté ce qui produira l'affichage

```
$A
```

Par contre

```
eval $COMMAND $ARGS
```

est récrit en

```
eval echo $A
```

qui provoque la réévaluation de

```
echo $A
```

et produit le résultat escompté.

5.9 Exemples

Voici pour terminer ce chapitre quelques exemples non commentés que le lecteur pourra étudier.

5.9.1 psg

Cette fonction recherche un processus par son nom.

psg

```
psg ()
{
    case "$UNIX" in

        SV )
                PSG_OPT=-ae
                ;;

        BSD )
                PSG_OPT=-ax
                ;;

        * )
                echo "$0: warning: UNIX variable unset" > /dev/tty
                ;;
    esac

    ps $PSG_OPT $2 | egrep -e $1\|PID | grep -v grep
}
```

psg

```
$ psg xfig
  PID TT STAT  TIME COMMAND
 8921 p1 IW    3:41 xfig
$
```

5.9.2 Xdisp

Cette fonction positionne de la variable `DISPLAY`.

Xdisp

```
Xdisp()
{
    if [ $# = 1 ]
    then
        DISP=$1
    else
        DISP=${DEFAULT_DISPLAY-`hostname`}
```

```
    fi

    echo Display on $DISP
    export DISPLAY=$DISP:0.0

    unset DISP
}
```

Xdisp

```
renoir: Xdisp
Display on renoir
renoir:
renoir: echo DISPLAY = $DISPLAY
DISPLAY = renoir:0.0
renoir:
renoir: Xdisp hitchcock
Display on hitchcock
renoir:
renoir: DEFAULT_DISPLAY=cassavetes
renoir: export DEFAULT_DISPLAY
renoir:
renoir: Xdisp
Display on cassavetes
renoir:
```

5.9.3 last

Cette commande affiche le noms des n fichiers le plus récemment modifiés.

last

```
#!/bin/sh

COMMAND_NAME=‘basename $0‘

usage ()
{
        echo Usage: $COMMAND_NAME [number_of_files] > /dev/tty
        exit 1
}

if [ $# = 0 ]
then
        N=-1
else
        case $1 in
            -* )
                N=$1
                ;;
            * )
                N=-$1
                ;;
        esac
fi
```

```
ls -1t $DIR | head $N || usage
```

last

```
$ last
ch-shell.tex
$ last 3
ch-shell.tex
ch-envtrav.tex
tmp.ps
$ touch main.tex
$ last
main.tex
$
```

5.9.4 makemake

Cette commande construit automatiquement un fichier Makefile pour des cas simples d'application.

makemake

```
#!/bin/sh

# Exemples d'utilisation
#
# 1 : makemake editeur
#     construction du fichier Makefile construisant un exe'cutable de nom
#     editeur a' partir des sources du re'pertoire courrant
#
# 2 : makemake -ltermcap -lm editeur
#     comme 1, avec utilisation des bibliothe'ques termcap et m lors de
#     l'e'dition de liens
#
# 3 : makemake -L.. -I../include -lgeneral editeur
#     comme 1, avec definition de repertoires include et bibliotheques

MAKEFILE=Makefile
BACKUP=.BAK
VARCC=gcc
PARSING="$VARCC -MM"
CMD=`basename $0`

usage()
{
        echo "Usage: $CMD [-I<path>...] [-L<path>...] [-l<lib>...] [<file>]"
        exit 1
}

makemake_backup ()
{
    if [ -f $MAKEFILE ]
    then
            mv $MAKEFILE $MAKEFILE$BACKUP
    fi
}

makemake_getargs ()
```

```
{
    while :
    do
        case $1 in

          -l* )
            VARLD="$VARLD $1"
            ;;

          -I* )
            CPPFLAGS="$CPPFLAGS $1"
            ;;

          -L* )
            LDDIRS="$LDDIRS $1"
            ;;

          -* )
            usage
            ;;

          * )
            break
            ;;
        esac
        shift
    done

    case $# in
        0)
            RESULT=a.out
            ;;
        1)
            RESULT=$1
            ;;
        *)
            usage
            ;;
    esac
}

makemake_make ()
{
    echo "CC= $VARCC"
    echo "CFLAGS= -g"
    echo "CPPFLAGS=$CPPFLAGS"
    echo "OUTFILE=$RESULT"
    echo OBJ= *.c | sed -e 's/\.c/\.o/g'
    echo
    echo '$(OUTFILE)' : '$(OBJ)'
    echo '      $(CC) $(CPPFLAGS) $(CFLAGS) $(OBJ)' $LDDIRS $VARLD '-o $@'
    echo
    $PARSING $CPPFLAGS *.c
    echo
    echo "clean :"
    echo '      rm -f $(OBJ) $(OUTFILE)'
}

makemake_getargs $*
makemake_backup
makemake_make > $MAKEFILE
```

```
$ ls
dline.c  dline.h  main.c   oc.c     oc.h
$ gcc -MM *.c
dline.o: dline.c dline.h oc.h
main.o: main.c dline.h oc.h
oc.o: oc.c oc.h
$
$ makemake dline
$ ls
Makefile  dline.c   dline.h   main.c    oc.c      oc.h
$
$ more Makefile
CC= gcc
CFLAGS= -g
CPPFLAGS=
OUTFILE=dline
OBJ= dline.o main.o oc.o

$(OUTFILE): $(OBJ)
        $(CC) $(CPPFLAGS) $(CFLAGS) $(OBJ) -o $
dline.o: dline.c dline.h oc.h
main.o: main.c dline.h oc.h
oc.o: oc.c oc.h

clean:
        rm -f $(OBJ) $(OUTFILE)
$
$ make
gcc -g   -c dline.c -o dline.o
gcc -g   -c main.c -o main.o
gcc -g   -c oc.c -o oc.o
gcc  -g dline.o main.o oc.o -o dline
$
$ ls
Makefile  dline.c   dline.o   main.o    oc.h
dline     dline.h   main.c    oc.c      oc.o
$
$ make clean
rm -f dline.o main.o oc.o dline
$
$ ls
Makefile  dline.c   dline.h   main.c    oc.c      oc.h
$
```

Exercices.

1. Empiler plusieurs shells: `csh`, puis `sh` et observer la parenté des différents processus par `ps -l`.
2. Écrire un script, pour l'interprète `sh`, qui enchaîne l'exécution de plusieurs commandes UNIX et le faire exécuter. Que faut-il faire pour qu l'exécution soit possible? Ajouter la ligne `set -x`, ou set -v dans le script. Quel est l'effet produit?
3. Afficher la valeur de `PATH`. Sauvegarder cette valeur dans une variable `PATHSVG` et vérifier la valeur sauvegardée. Mettre la chaîne vide dans `PATH` et essayer `cd`, `pwd`, `ls`. Que peut on en conclure? Comment

lancer `ls` sans modifier `PATH`? Modifier `HOME` et observer ce que produit `cd` sans argument. Restaurer les valeurs de `PATH` et de `HOME`.

4. Observer la valeur `$?` retournée par l'exécution d'une commande. Tester les deux cas possibles: terminaison normale et terminaison anormale.
5. Tester des enchaînements de commandes au moyen des connecteurs `&&` et `||`.
6. Faire afficher les chaînes suivantes
   - `***␣BONJOUR␣***`
   - `␣-␣␣␣Le␣caractere␣'*'`
   - `/Fichier␣d'entree\`
   - `*␣Mon␣Jan␣20␣17:59:48␣EET␣1992␣*` (la date doit être le résultat d'une exécution de `date`)
   - `$$␣Mon␣Jan␣20␣18:07:15␣EET␣1992␣$$` (idem)
   - `$6248$` (où le nombre entre `$` est le *pid* du *shell*)
   - `(␣/bin/adb␣/bin/cat␣/bin/rcp␣/bin/tar␣/bin/vax␣)` (utiliser un remplacement de chemins).
7. Écrire un script qui affiche l'heure sous la forme:

   `Il est 17:45:26.`
8. En modifiant la variable `IFS` qui contient la liste des séparateurs, écrire un script qui affiche:

   `Il est 17h 45mn.`
9. Modifier son environnement de façon à ce que ces commandes soient utilisables depuis n'importe quel répertoire (on créera un répertoire `bin` dans son répertoire d'accueil et on modifiera la variable `PATH`).
10. Rendre la modification de la variable `PATH` valide d'une connexion sur l'autre.
11. Écrire, sans utiliser de structure de contrôle, les commandes suivantes:
    - `nf` : affichage du nombre de fichiers du répertoire courant;
    - `ra` : affiche `oui` si le répertoire courant est le répertoire d'accueil et `non` sinon;
    - `prc` : affiche la profondeur du répertoire courant;
    - `ouest` : `ouest` ⟨*nom*⟩ affiche

      ⟨*nom*⟩ `est dans` ⟨*rep*⟩

      où ⟨*rep*⟩ est le répertoire d'accueil de ⟨*nom*⟩ si ⟨*nom*⟩ est un utilisateur défini, et

      ⟨*nom*⟩ `n'existe pas`

      sinon.
12. Écrire le script `ifdef` tel que l'exécution de `ifdef` ⟨*nom*⟩`.h` place en tête de ⟨*nom*⟩`.h` les lignes

    `#ifndef` ⟨*NOM*⟩`_H`
    `#define` ⟨*NOM*⟩`_H`

    et la ligne

    `#endif /*` ⟨*NOM*⟩`_H */`

    à la fin. On pourra utiliser la commande `tr(1)` pour les conversions de minuscules en majuscules.
13. Expliquer le comportement de chacune des lignes de commandes suivantes:

```
A='echo *'
echo '$A'
echo "$A"
echo $A
$A
```

14. En observant le comportement des lignes de commande suivantes, expliquer ce que fait `eval`.

```
LIG='$CMD [$ARG]'
CMD=echo
ARG='$$'
$LIG
eval $LIG
eval eval $LIG
eval eval eval $LIG
```

15. Tester l'utilisation de variables pour définir des routines:
    - `USAGE` : affichage d'un message d'usage et provoquant une terminaison;
    - `ERREUR` : affichage d'un message d'erreur paramétré par une variable `MES_ERR`).

16. Observer et expliquer la suite de commandes:

```
echo $A
A=aaaaa
echo $A
sh
echo $A
C-d
export A
sh
echo $A
A=bbbbb
echo $A
C-d
echo $A
```

17. Enlever et remettre l'attribut *export* à la variable `PS1` et vérifier à chaque fois le résultat en empilant un *shell*.

18. Que fait le script suivant:

```
#!/bin/sh

if [ $# = 0 ]
then
    cd ..
else
    cd $1
fi
echo " --> `pwd`"
```

Tester cette commande et expliquer ce qui se produit à l'exécution. Essayer en préfixant la commande par `.` ou par `source`. Que peut-on en conclure?

19. En utilisant la commande `test` et la structure de contrôle `if` modifier la commande `ifdef` de manière à

- faire afficher un message d'usage en cas d'utilisation sans argument;
- créer un fichier ⟨*nom*⟩`.h` dans le cas où il n'existe pas;
- ne pas modifier le fichier s'il commence déjà par un ligne `#ifndef` ⟨*NOM*⟩`_H`.

20. Tester la boucle `for` au moyen du programme suivant:

```
for N in un deux trois quatre
do
   echo "**$N**"
done
```

Tester la boucle `while` et la commande `shift`.

21. Modifier la commande `ifdef` de manière à ce qu'elle traite une liste de fichier. Vérifier en faisant exécuter plusieurs `ifdef *.h`.

22. Écrire un script `backup` tel que `backup` ⟨*nom*⟩ `...` recopie le fichier ⟨nom_i⟩ en ⟨nom_i⟩`.OLD`. L'option `-s` permet de choisir le suffixe; l'option `-b` produit une sauvegarde de la sauvegarde si elle existe déjà.

23. Modifier `ifdef` de façon à ce qu'il

- accepte l'option `-C`: ajout du commentaire

  `/* Fichier` ⟨*nom*⟩`.h */`

- rajoute automatiquement le suffixe `.h` s'il n'est pas présent dans ⟨*nom*⟩.

Annexe A

Exercices et problèmes d'examen

Les exercices et problèmes listés dans cette annexe sont issus des sujets d'examen de janvier 1992 à septembre 1995.

1. Expliquer la différence entre le stoppage d'un processus et sa terminaison; donner deux exemples de stoppage et de réveil de processus.
2. Expliquer comment UNIX traite un chemin relatif et comment il traite un chemin absolu. Donner deux cas d'erreur pouvant se produire lors du traitement d'un chemin.
3. Quelle est la différence, sous *GNU Emacs*, entre une fenêtre, un buffer, et un fichier? Peut-on avoir:

   a) un buffer sans fichier?
   b) un fichier sans buffer?
   c) une fenêtre sans buffer?
   d) un buffer sans fenêtre?

4. Expliquer l'effet produit sous *GNU Emacs* par

   a) `C-x ( C-a / * C-e * / C-x )`
   b) `M-x name-last-kbd-macro comment-line`
   c) `local-set-key C-c / comment-line`

   Comment peut on vérifier que l'on ne détruit pas une fonction ou une liaison déjà définie.
5. En *shell*, donner un exemple d'utilisation de la valeur d'une commande, et un exemple de l'utilisation du résultat d'une commande.
6. Le but de cette question est d'illustrer le fonctionnement général d'un *shell*:
   - saisie d'une ligne de commande,
   - résolution des substitutions,
   - découpage en mots et évaluation de la commande.

   On suppose que l'on est dans un répertoire contenant les fichiers de nom

   ```
   ls    mv    x1    x2
   ```

   Expliquer le comportement d'un *shell* et le résultat obtenu lorsque l'on entre successivement les lignes de commande suivantes:

   ```
   echo *
   *
   `*`
   echo *
   ```

Remarque. ***Les caractères entourant l'étoile sur la troisième ligne sont des accents graves (*backquotes*).***

7. Écrire au moyen d'un script *shell* la commande `rep` décrite par la page de manuel suivante:

NAME

rep - affiche une liste de répertoires

SYNOPSIS

```
rep nomdefichier ...
```

DESCRIPTION

Pour chaque nom *nomdefichier* reçu en argument, s'il s'agit d'un répertoire, la commande *rep* l'affiche sur sa sortie standard.

EXAMPLE

Par exemple l'exécution de la commande

```
rep /etc/*
```

produit la liste

```
/etc/adm
/etc/install
/etc/spool
/etc/tmp
```

SEE ALSO

– sh(1), test(1)

8. En utilisant des commandes UNIX standard et les mécanismes de redirection, écrire une ligne de commande *shell* qui affiche le numéro de processus (*pid*) de toutes les occurrences de *bash* présentes en machine.
9. Lorsqu'on écrit une nouvelle commande UNIX, que doit on faire pour l'installer correctement dans le système (fichier exécutable, page de manuel, ...).
10. Écrire une macro *GNU Emacs* qui recopie une ligne sur la ligne suivante, en plaçant la première lettre à la fin de la ligne. Par exemple, si l'on est dans la situation suivante:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
□
```

où □ représente le curseur et où la lettre `A` est située sur la première colonne, l'exécution de la macro produit

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
BCDEFGHIJKLMNOPQRSTUVWXYZA
□
```

Au moyen de cette macro, comment obtenir en une commande, à partir de la première ligne, le

texte suivant?

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
BCDEFGHIJKLMNOPQRSTUVWXYZA
CDEFGHIJKLMNOPQRSTUVWXYZAB
DEFGHIJKLMNOPQRSTUVWXYZABC
                .
                .
                .
XYZABCDEFGHIJKLMNOPQRSTUVW
YZABCDEFGHIJKLMNOPQRSTUVWX
ZABCDEFGHIJKLMNOPQRSTUVWXY
```

11. Expliquer le fonctionnement de la fonction *GNU Emacs* suivante

```
(defun goto-column (c)
  (let ((d (dot)))
    (beginning-of-line)
    (untabify (dot) d))
  (forward-char c))
```

12. Écrire une commande *shell* qui parcourt tous les répertoires dont le nom est passé en argument, et qui supprime tous les fichiers *nom*~ pour lesquels le fichier *nom* existe. Sans arguments, le travail est effectué dans le répertoire courant.
13. Modifier la commande de la question précédente de façon à ce que l'option `-R` provoque le parcourt récursif de chaque arborescence.
14. Comment peut-on écrire une commande *shell* qui empêche la redirection de son entrée standard? Comment peut-on le faire avec une commande écrite en C?
15. Quelles différences y a-t-il entre la sortie standard et la sortie standard erreur? Donner des exemples d'utilisation.
16. Donner quatre cas distincts d'erreur pouvant se produire lors de l'exécution sous bash de la commande

```
$ mv a.out bin/kl
```

17. Dans la session suivante, donner deux valeurs possibles pour chacune des options $\langle option_1 \rangle$ à $\langle option_4 \rangle$.

```
$ ls -l cmd
-rw-r--r-- 1 root      0 May 17 16:17 cmd
$ chmod ⟨option1⟩ cmd
$ ls -l cmd
-rwxr-xr-x 1 root      0 May 17 16:17 cmd
$ chmod ⟨option2⟩ cmd
$ ls -l cmd
-rwxr-xr-- 1 root      0 May 17 16:17 cmd
$ chmod ⟨option3⟩ cmd
$ ls -l cmd
-rwx--x--- 1 root      0 May 17 16:17 cmd
$ chmod ⟨option4⟩ cmd
$ ls -l cmd
-rws--x--- 1 root      0 May 17 16:17 cmd
$
```

18. Quelle est, sous *GNU Emacs*, la différence entre une clé, une commande et une fonction. Donner des exemples.
19. On suppose que les lignes suivantes sont tapées depuis une session sous un shell, par exemple *bash*:

    ligne 1: `emacs main.c` ⟨*RETURN*⟩
    ligne 2: `M-x suspend-emacs`
    ligne 3: `emacs main.c` ⟨*RETURN*⟩
    ligne 4: `C-k C-k C-x C-s`
    ligne 5: `C-x C-c`
    ligne 6: `jobs` ⟨*RETURN*⟩
    ligne 7: `fg` ⟨*RETURN*⟩
    ligne 8: `C-d C-d C-x C-s C-x C-c`

    On peut remarquer qu'il s'agit là d'une utilisation incorrecte de *GNU Emacs*, qui peut entraîner des erreurs lors de la manipulation du fichier `main.c` (on suppose que le fichier `main.c` existe dans le répertoire courant et qu'il est non vide).
    Indiquer, pour chaque ligne:
    - le logiciel qui la traite;
    - le résultat produit;
    - quand il y a lieu, la ou les erreurs qu'elle peut produire.
20. Écrire en shell la commande `clean` dont l'utilisation est la suivante:

    `clean [-i] [`⟨*name*⟩`...]`

    Cette commande détruit dans chaque répertoire de la liste ⟨*name*⟩... les fichiers *back-up* de *GNU Emacs* et `core`. Si aucun répertoire n'est précisé, le travail est effectué dans le répertoire courant.
    L'option `-i` provoque une demande de confirmation interactive pour chaque suppression.
21. Donner six appels système utilisés par le shell, avec pour chacun, une situation de son utilisation. Par exemple:

    - `signal`: utilisé pour programmer la commande `trap` du shell.

    Remarque. Attention à ne pas confondre le shell avec les autres commandes du système. Par exemple:
    - `unlink`: utilisé dans le code de la commande `rm`

    est une mauvaise réponse, car `rm` n'est pas une commande interne du shell.
22. Comment calcule-t-on le code ASCII des caractères `C-`⟨*c*⟩ et `M-`⟨*c*⟩ à partir de celui du caractère ⟨*c*⟩ ? Les combinaisons CONTROL et META sont-elles utilisables simultanément ?
23. On considère, sous `bash`, un environnement dans lequel l'exécution de la ligne de commande

    `mkdir tmp ; pwd ; ls -ld tmp`

    produit le résultat

    ```
    ~/tmp
    drwxr-xr-x  2 modele       512 Aug 31 12:38 tmp
    ```

    Donner, en l'expliquant, le résultat produit par chacune des lignes commandes suivantes.
    a) `chmod u-r tmp ; ls tmp`
    b) `cd tmp ; pwd`
    c) `echo abcd >f ; ls`
    d) `cd .. ; chmod u+r tmp ; ls tmp`

e) `chmod u-x tmp ; ls tmp ; cd tmp`

f) `chmod u+x-w tmp ; rm tmp/f`

g) `echo abcdefgh >tmp/g`

h) `echo efgh >> tmp/f ; cat tmp/f`

24. Expliquer ce que fait, sous *GNU Emacs*, la fonction `f` suivante :

```
(setq f-chaine "|> ")

(defun f ()
  (interactive)
  (if (> (dot) (mark))
    (exchange-dot-and-mark))
  (while (< (dot) (mark))
    (beginning-of-line 1)
    (insert f-chaine)
    (next-line 1)))
```

Donner un exemple de son utilisation. Voici pour information les messages affichés sous *GNU Emacs* dans le *buffer* `*Help*` par la fonction `describe-function` pour les fonctions `dot`, `mark`, `insert`, `beginning-of-line`, `exchange-dot-and-mark` et `next-line`.)

`point`: Return value of point, as an integer. Beginning of buffer is position (point-min)

`mark`: Return this buffer's mark value as integer, or nil if no mark. If you are using this in an editing command, you are most likely making a mistake; see the documentation of set-mark

`exchange-dot-and-mark`: Put the mark where point is now, and point where the mark is now.

`beginning-of-line`: Move point to beginning of current line. With argument ARG not nil or 1, move forward ARG - 1 lines first. If scan reaches end of buffer, stop there without error.

`insert`: Any number of args, strings or chars. Insert them after point, moving point forward.

`next-line`: Move cursor vertically down ARG lines. If there is no character in the target line exactly under the current column, the cursor is positioned after the character in that line which spans this column, or at the end of the line if it is not long enough. If there is no line in the buffer after this one, a newline character is inserted to create a line and the cursor moves to that line.

25. Écrire, en au plus une dizaine de 10 lignes de shell, la commande `itere` telle que l'exécution de

`itere` $\langle n \rangle$ $\langle arg_0 \rangle$ $\langle arg_1 \rangle$ ...

itère $\langle n \rangle$ fois l'exécution de la ligne de commande $\langle arg_0 \rangle$ $\langle arg_1 \rangle$ ... Voici des exemples de son exécution :

```
$ itere 3 date
Tue Aug 31 13:48:38 MET DST 1993
Tue Aug 31 13:48:39 MET DST 1993
Tue Aug 31 13:48:39 MET DST 1993
$
$ itere 4  echo + + + +
+ + + +
+ + + +
+ + + +
+ + + +
$
```

On utilisera les commandes `expr` et `test`.

26. Voici une ligne de commande du *shell* standard du système UNIX :

```
for N in `cat /etc/passwd | sed -e 's/:.*//'` ; do ( echo Test mail \
  | mail $N && echo $N: OK ) >> MAIL.LOG ; done
```

a) Donner la liste des commandes simples exécutées par sh lors du traitement de cette ligne de commande.

b) Décrire ce traitement en détaillant le résultat et la valeur de chaque commande simple, les redirections, *pipes*, etc.

Remarque. La commande `sed -e s/`m_1`/`m_2`/` remplace dans chaque ligne du flot d'entrée standard une occurrence du motif m_1 par le motif m_2. Le motif `.*` code une suite maximale de caractères différents de `\n`).

c) Citer six appels système utilisés par le *shell* lors de l'évaluation de cette commande, et expliquer pour chacun des cas en quelques mots la raison de son utilisation.

27. On suppose qu'à la suite d'une erreur du super-utilisateur, la date du système a été retardée d'un jour.

a) Expliquer ce que cela peut entrainer comme désagrément pour un développeur lorsqu'il reconstruit son application.

b) Écrire en *shell* standard un script qui supprime tous les fichiers suffixés par `.o` dont la date de dernière modification est celle du jour suivant. Par exemple, si la date courante est le 24 mai, cette commande supprime tous les fichiers `*.o` du 25 mai. On donnera une version simplifiée qui ne traite pas le cas du dernier jour du mois. On rappelle sur un exemple les formats par défaut des résultats de `ls` et de `date` :

```
-rw-r--r--  1 achille      3808 Aug 31  1993 juin-93.tex
-rw-r--r--  1 achille     11919 May 23 22:17 juin-94.tex.crypt

Tue May 24 09:05:16 MET DST 1994
```

Le septième champ du résultat de `ls` est l'année si elle est différente de l'année courante, et l'heure sinon.

28. Donner la marche à suivre sous *GNU Emacs* pour définir une macro-clavier remplaçant une fonction C par son prototype. Par exemple, la fonction

```
int pile_vide(pile p)
{
    return p->sommet != p->base;
}
```

est transformée en

```
extern int pile_vide(pile p);
```

Cette macro devra pouvoir être itérée pour être appliquée à tout le fichier.

Remarque. On rappelle que la fonction `forward-list` (resp. `backward-list`) liée à la clé `M-C-n` (resp. `M-C-b`) positionne le curseur à la fin (resp. au début) du système de parenthèses ou d'accolades suivant (resp. précédant)).

29. Peut-on rendre impossible la suppression d'un fichier tout en permettant celle des autres fichiers du même répertoire (si oui comment, si non pourquoi) ?

30. La commande `time` fait exécuter une commande dont elle reçoit le nom en argument, puis lorsque cette commande se termine, affiche le temps de calcul consommé par le système, le temps de calcul consommé par la commande, et le temps réel qui s'est écoulé. Par exemple, dans l'extrait de session

```
$ time latex septembre-94.tex
[1] [2]
Transcript written on septembre-94.log.
        5.1 real         1.0 user         0.3 sys
$
```

la commande `time` lance la commande `latex septembre-94.tex` qui affiche les deux lignes suivantes. Puis la commande `time` affiche la ligne

```
        5.1 real         1.0 user         0.3 sys
```

signifiant que l'exécution de la commande `latex` a consommé 1 seconde de temps UC (Unité Centrale) et que le noyau UNIX a durant cette même exécution consommé 0.3 secondes de temps UC. Cette exécution s'est déroulée pendant 5 secondes de temps réel. Expliquer ce que fait le noyau pendant ces 0.3 secondes et pourquoi la somme des temps UC utilisateur et système est très inférieure au temps réel.

31. Sous *GNU Emacs*, donner 4 exemples de buffers créés implicitement en expliquant dans quelles circonstances ils sont créés.
32. Les buffers de la question précédente ne sont associés à aucun fichier. Comment faire pour associer un tel buffer à un fichier?
33. Écrire une commande *shell* qui, étant donnée une liste $\boldsymbol{L} = N_1, N_2, \ldots, N_p$ de noms de fichiers recopie chaque fichier N_i dans un fichier de nom $.N_i.aammjj\text{-}hhMM$ où aa, mm, jj, HH, MM sont des nombres de deux chiffres dont la valeur est respectivement l'année courante, le numéro du mois et du jour, et l'heure courante (heures et minutes). Par exemple si la commande `date` affiche

    ```
    Wed Sep 7 09:52:29 MET DST 1994
    ```

    chaque nom de fichier créé sera suffixé par

    ```
    .940907-0952
    ```

    Si la commande est appelée sans arguments, la liste $\boldsymbol{L}$ est la liste des fichiers du répertoire courant, et sinon c'est la liste des arguments de la commande. Si un nom de la liste est un nom de répertoire, il est ignoré. La commande vérifiera également que chaque nom à traiter est un nom simple, c'est-à-dire qu'il ne contient pas de caractère `/`. La page de manuel de la commande `date` est donnée en annexe du devoir.
34. On considère sous UNIX la session *shell* suivante:

    ```
    $ ls -l tmp
    -r--r--r--    1 npuvyyr  1108       12328 May 16 15:16 tmp
    $ rm tmp
    tmp: 444 mode. Remove ? (yes/no)[no] : yes
    $ ls -l tmp
    Cannot access tmp: No such file or directory
    $
    ```

    a) Expliquer la raison d'être du message produit par l'exécution de la commande `rm`.

    b) La suppression du fichier `tmp` est-elle effectuée et pourquoi?

35. On considère la ligne de commande *shell*

    ```
    mail `cat mlist | grep JURY | sed -e /JURY//` <jurys.message
    ```

    a) Expliquer le déroulement de cette commande et décrire l'effet produit, en détaillant les différentes étapes de son exécution par un *shell*.

    b) Décrire l'enchaînement des appels système mis en oeuvre par un *shell* lors de cette exécution. Pour chaque appel utilisé, on décrira brièvement ce qu'il fait et la raison précise de son utilisation dans le cadre de cette exécution. On demande de donner au moins 5 cas distincts et non triviaux (autres que `read` et `write`).

36. On considère un fichier contenant une image noir et blanc dont le format est le suivant :

$$\begin{array}{l} \Delta_x \\ \Delta_y \\ g_1 \ \cdots g_{\Delta_x} \\ \vdots \\ g_{(\Delta_x \times \Delta_y - 1)+1} \ \cdots g_{\Delta_x \times \Delta_y} \end{array}$$

où Δ_x et Δ_y sont deux entiers et où chaque g_i code le niveau de gris d'un pixel. Sur chaque ligne, les valeurs g_i sont séparées par une ou plusieurs espaces.

a) On a besoin de connaître la valeur maximale de niveau de gris de l'image. Décrire une suite d'opérations, effectuée sous *GNU Emacs*, et basée sur l'utilisation de la commande

`sort-numeric-fields` [1]

permettant de connaître cette valeur.

b) Écrire un *pipeline* en *shell* prenant le fichier image sur son flot d'entrée et produisant pour résultat la valeur maximale de niveau de gris [2].

37. Écrire au moyen d'un *shell script* la commande UNIX `pfind` dont la syntaxe est la suivante:

`pfind` ***pathlist name...***

où ***pathlist*** est une liste de répertoire au format de la variable d'environnement `PATH`, et ***name...*** est le nom d'un fichier à rechercher. La commande recherche le fichier dans tous les répertoires spécifiés dans ***pathlist*** et affiche le chemin complet de chaque occurrence trouvée. Exemple :

```
$ echo $TEXINPUTS
.:/usr/labri/achille/lib/tex/inputs:/usr/local/lib/tex/inputs:/usr/labri/achille/
lib/tex/sty
$
$ pfind $TEXINPUTS farticle.sty
/usr/labri/achille/lib/tex/inputs/a4.sty
/usr/local/lib/tex/inputs/a4.sty
$
```

38. Décrire trois cas d'erreur pouvant se produire lors de l'exécution de l'appel système `unlink` de destruction de fichier.

39. Écrire au moyen d'un *shell script* la commande `del` dont la syntaxe est

`del` $\langle fichier \rangle$`...`

et qui place chaque fichier de la liste $\langle fichier \rangle$`...` dans un répertoire de nom `.to_del` situé dans le répertoire d'accueil de l'utilisateur. S'il existe déjà un fichier de même nom, par exemple `xxx`, dans le répertoire `.to_del`, le fichier déplacé est renommé `xxx~1`. S'il existe également un fichier de nom `xxx~1`, le fichier déplacé est renommé `xxx~2`, et ainsi de suite.

40. Sous *GNU Emacs*, il est possible de se retrouver avec deux buffers distincts ouvert sur un même fichier.

a) Décrire une suite d'actions aboutissant à ce cas de figure.

b) Quel danger y a-t-il à se trouver dans une telle configuration ?

c) Lorsque cela se produit, est-ce qu'on peut facilement s'en rendre compte ?

---

1. Par défaut, cette commande trie les lignes de la région courante par ordre croissant, en considérant la valeur numérique des suites de chiffres – par exemple la chaîne `10` est supérieure à la chaîne `9`, ce qui n'est pas vrai pour l'ordre lexicographique.

2. Indication : l'option `-n` de la commande `sort` permet l'utilisation de l'ordre numérique à la place de l'ordre lexicographique.