ABC d'Unix

Christian Queinnec
Université Paris 6 — Pierre et Marie Curie
LIP6, 4 place Jussieu, 75252 Paris Cedex
France – Email: `Christian.Queinnec@lip6.fr`

Revision: 2.10

Cet ouvrage a été réalisé dans le cadre d'un cours professé à l'École Nationale Supérieure des Techniques Avancées vers 1984-1986. Il a été édité chez Eyrolles en 1985 mais est, depuis longtemps, introuvable. Il recouvre maintenant une seconde jeunesse, sur le réseau, et, bien sûr, sur le cédérom VideoC. Il était initialement écrit en nroff, je l'ai converti en LaTeX (avec un script en Perl et des flopées de macros Emacs). J'ai également corrigé quelques fautes d'orthographe et tenté de récupérer les dessins originellement créés en MacPaint. Le processus de traduction a dû introduire de nouvelles erreurs que votre sagacité réactive ne manquera pas de me signaler.

Tout cela a un petit goût des années 1980 (j'ai cependant supprimé deux annexes particulièrement obsolètes) mais comme la connaissance de **sh** n'a pas vraiment percé depuis, ce petit ouvrage a encore de l'avenir.

Paris 1985 — Paris 1999, rev. 2000.

[1] http://www.gnu.org/copyleft/fdl.shtml
[2] http://youpou.lip6.fr/queinnec/Books/ABCdUNIX/uunix-toc.html
[3] http://youpou.lip6.fr/queinnec/Books/ABCdUNIX/uunix.ps.gz

Chapitre 1

Prologue

1.1 Introduction

UNIX est un système de production de programme. Il a lui-même profité de ses nombreuses qualités aussi est-il remarquablement documenté. Toutefois cette documentation, que plus de dix années d'utilisation ont largement éprouvée, peut, à première vue, souffrir de deux défauts :

- Elle est en américain ;
- Elle est extrêmement concise ! Tout y est dit, rien n'y est répété et tout ce qui est omis est, en général, faux ! ! !

Afin de n'être ni une pâle traduction ni une logorrhéïque paraphrase, nous avons pris le parti de ne présenter que les éléments qui vous permettront de pénétrer de plein pied dans le monde d'essaiera-t'on de dégager les principaux concepts de sa *philosophie* afin que se cristallise en votre esprit un modèle adéquat de ce merveilleux système. L'essence occulte les détails et seuls ceux caractéristiques d'UNIX trouveront ici place. Ne cherchez donc pas tout dans cet ouvrage ! Les réponses à vos questions se trouvent dans la documentation complète *UNIX Programmer's Manual, Volume 1 et suivants* qui existe probablement en de nombreux exemplaires dans votre bibliothèque ou dans les parages de votre centre de calcul. Si vos besoins vous l'imposent, exigez la ! ! ! S'en priver serait en effet vous exposer à des travaux fastidieux et superflus. L'apprentissage efficace d'un programme, quelqu'il soit et aussi soigné soit-il, ne saurait s'acquérir sans documentation ; si quelques essais relatifs à un objet précis lèvent un doute, ils ne peuvent à eux seuls la remplacer.

L'ouvrage abordera successivement les objets d'UNIX, les principales commandes (mais sans détailler leurs multiples options : consulter pour cela la documentation de référence), le système de gestion des fichiers et enfin l'interface avec l'utilisateur qui nous a toujours semblé négligée dans ce genre d'ouvrage, qui est un exemple intéressant du concept des couches logicielles et qui, enfin, est responsable de tant de déboires et d'incompréhension dans le maniement d'UNIX.

UNIX est un tout que ce livre s'efforce de présenter linéairement. Certains détails ou exemples ne se laisseront comprendre qu'après une assimilation superficielle de l'ensemble. Ils ont néammoins été inclus car de nombreux points d'UNIX sont classiques parmi les systèmes d'exploitation et un exemple d'emploi renseigne alors plus que de verbeuses explications. Ils constituent autant de prétextes à une exploration plus approfondie, voire même plus expérimentale, du manuel de référence, du système d'exploitation et de ses utilitaires.

Chapitre 2

Rudiments

2.1 Introduction

UNIX est né au début des années 1970, créé par Ken THOMPSON et Dennis RITCHIE. Quoique donc déjà assez vieux, ses qualités le font considérer comme moderne. Il lui aura fallu près de dix ans pour qu'il se répande dans toutes les industries du Nouveau et de l'Ancien Monde. UNIX est maintenant un standard de facto : c'est le système le plus reproduit au monde (en nombre d'incarnations sur différents types d'ordinateurs). Il est considéré comme LE système, par excellence, de développement de programmes.

Le terme UNIX possède plusieurs significations complémentaires mais ne saurait se borner à l'une de celles là.

- C'est tout d'abord un système d'exploitation multitâches et multi-utilisateurs conçu pour miniordinateurs.
- C'est une large collection de programmes utiles (des « utilitaires » au plein sens du terme) obtenu par sélection quasi-naturelle (des statistiques d'emploi ont permis d'écarter les programmes boudés des utilisateurs. Par ailleurs, avant de vous lancer dans la programmation spécifique d'un outil logiciel quelconque, relisez la documentation : il y a peu d'outils généraux qu'UNIX ne procure pas sous une forme ou une autre !)
- C'est enfin, parmi ces derniers, des interprètes de commandes (***sh**).

UNIX est très lié au langage C qui a servi à sa programmation quasi-complète. La seule existence d'un compilateur C sur une machine permet d'envisager sereinement le transport entier d' UNIX sur cette même machine : l'effort est donc réduit, ce qui incite de nombreux constructeurs à n'offrir qu' UNIX comme système d'exploitation.

Un système d'exploitation est un logiciel capable de mettre en œuvre un ordinateur et ses périphériques pour y créer, exécuter, manipuler et archiver des programmes ou des données. UNIX est multitâches : il est capable de mener de front plusieurs tâches à la fois, c'est-à-dire d'exécuter plusieurs programmes en quasi-simultanéité pour les utilisateurs (qui n'ont du temps qu'une perception imprécise) mais séquentiellement, pour lui, pendant de courts laps de temps de l'ordre d'une fraction de seconde. Le noyau (ou cœur d') UNIX, ainsi nomme-t'on cette partie réduite au seul système d'exploitation, peut être considéré comme un ensemble de sous-programmes fortement liés répondant à diverses fonctionnalités. Les « UNIX-like » ne sont que des noyaux différents procurant des interfaces d'appels plus ou moins équivalentes.

2.2 Les objets d'UNIX

UNIX distingue des flux, des tâches et des fichiers. Un fichier (*file*) est une suite ordonnée d'octets. La taille maximale des fichiers est bien sûr limitée par la taille physique des mémoires qui les retiennent. Un fichier peut contenir n'importe quelle séquence d'octets et en particulier

- des caractères formant un texte (comme cette documentation),
- des programmes écrits dans un certain langage (FORTRAN, Lisp, PROLOG, Pascal, C ...) ; ce sont aussi des textes.
- des instructions binaires auquel cas ce fichier représente un programme exécutable.

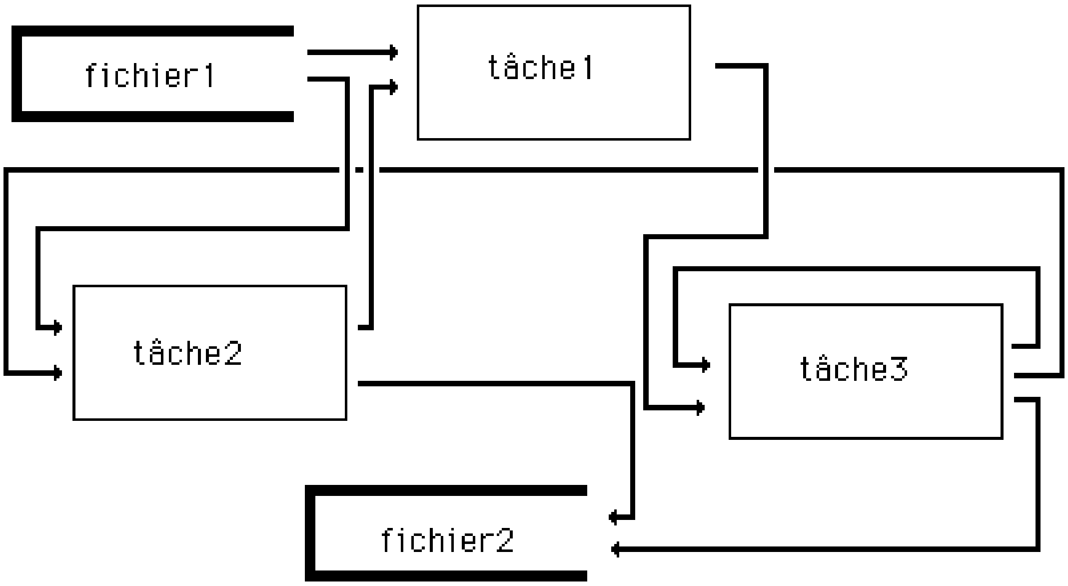
Un fichier peut être vide, c'est-à-dire exister et ne rien contenir !

Une tâche (*process*) est un programme en cours d'exécution. Un même programme peut être simultanément exécuté par plusieurs tâches : par exemple, plusieurs personnes peuvent user en même temps d'un même compilateur ou d'un même éditeur de texte. Une tâche naît, vit puis meurt. Une tâche peut même changer de programme support (cf. **exec**) sans pour cela devoir perdre la vie.

Une tâche effectue un certain travail en exploitant des flux (*stream*) pour en produire d'autres.

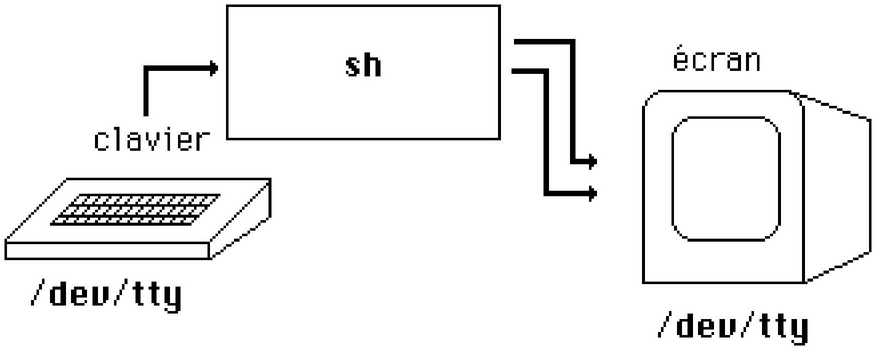
Un flux est une suite d'octets produite par un fichier ou une tâche en destination d'un fichier ou d'une tâche.

Tout agencement particulier de tâches, flux ou fichiers peut être mis en œuvre par UNIX, notamment des choses aussi tératologiques que :



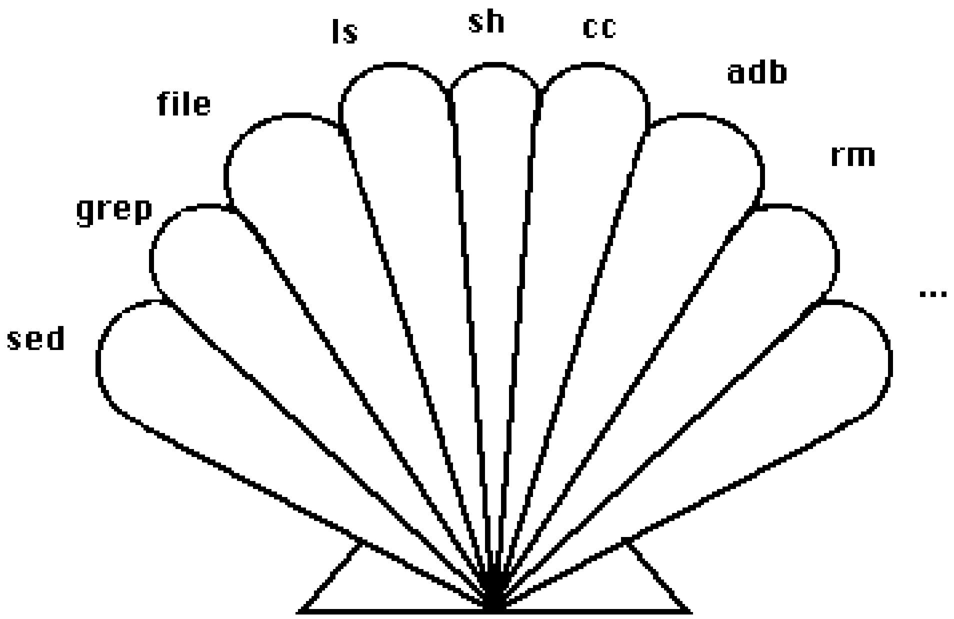
Quelques fichiers spéciaux existent qui mettent en œuvre les périphériques. Ainsi **/dev/tty** est-il, pour chaque utilisateur, son ensemble clavier-écran : lire **/dev/tty** permet de lire les caractères frappés ; écrire sur **/dev/tty** permet inversement d'afficher des textes sur l'écran.

Lorsque le calculateur est mis sous tension, un processus d'amorçage (*bootstrap*) se déroule et met en place pour chaque utilisateur potentiel une tâche particulière associée à trois flux connectés à un fichier spécial : **/dev/tty**.



Le rôle de cette tâche particulière est de permettre, après connexion et divers autres réglages, la mise en œuvre du noyau d'UNIX, c'est-à-dire la réalisation de schémas associant flux, fichiers et tâches. Elle lit les commandes frappées au clavier décrivant un schéma, les interprète c'est-à-dire détermine la suite des appels au noyau UNIX nécessaire pour former et lancer l'exécution du schéma souhaité (cf. « Presque le fin mot »).

Cette tâche est généralement une incarnation du programme **/bin/sh** et est connue sous le nom d'interprète de commandes ou *shell*. Le nom de *shell* (coquillage mais l'idée correspond mieux à celle d'un éventail) correspond à l'idée graphique suivante :



L'utilisateur se voit offrir un choix équanime entre tous les programmes accessibles. Tous les schémas concevables ne sont pas descriptibles en **/bin/sh**, mais la plupart des plus usuels le peuvent être.

2.3 Maniement en tout genre

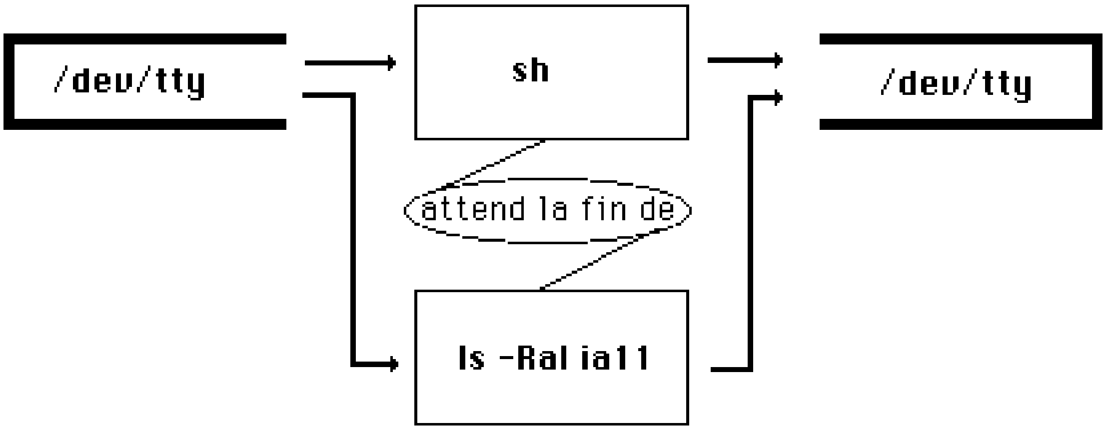
Une tâche est créée comme un appel à une fonction (le nom du programme sur lequel est créée la tâche) éventuellement suivie de ses arguments d'appel (les option d'exécution, le nom des fichiers concernés, etc ...). Par exemple :

```
ls  -Ral  ia11
```

est une commande de trois mots invoquant, c'est-à-dire demandant la mise en place d'un schéma où sera exécuté le programme **ls** (pour « lister ») avec deux arguments :

– les options **l**, **a** et **R** (le style UNIX veut que les option soient le plus souvent précédées d'un tiret)
– un nom : **ia11**

Lorsque une tâche est créée, elle hérite des flux connectés à sa mère (ie. la tâche qui l'a engendrée). Ainsi pour l'exemple précédent aura-t'on :

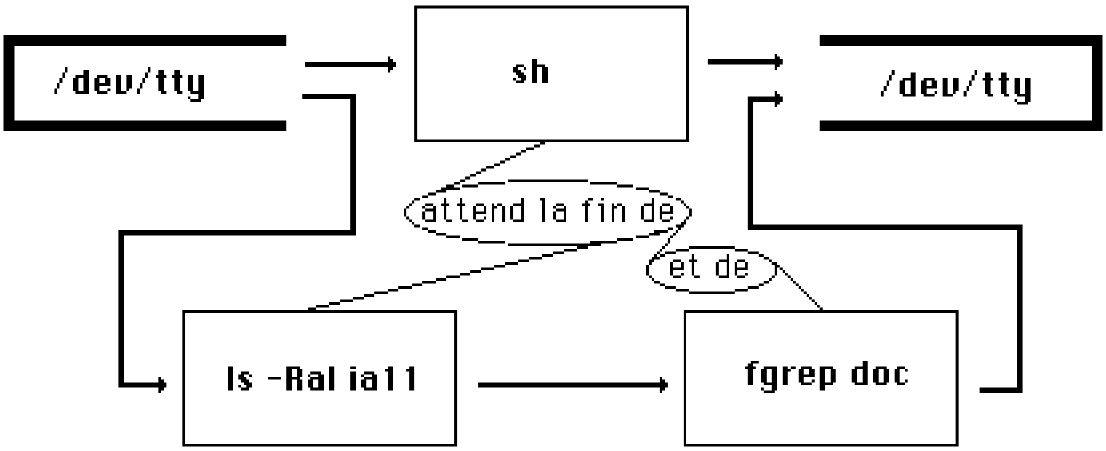


Parce qu'elles se partagent en lecture le clavier, la tâche mère est usuellement mise en sommeil le temps que la tâche fille s'achève. Lorsque la tâche de listage sera terminée, l'interprète de commandes sollicitera une nouvelle commande à partir du clavier.

La fameuse cascade (*pipe*) d'UNIX permet de connecter des tâches via un flux. Ainsi

```
ls -Ral ia11 | fgrep doc
```

conduit-il au schéma suivant où **ls** et **fgrep** sont mis en cascade.

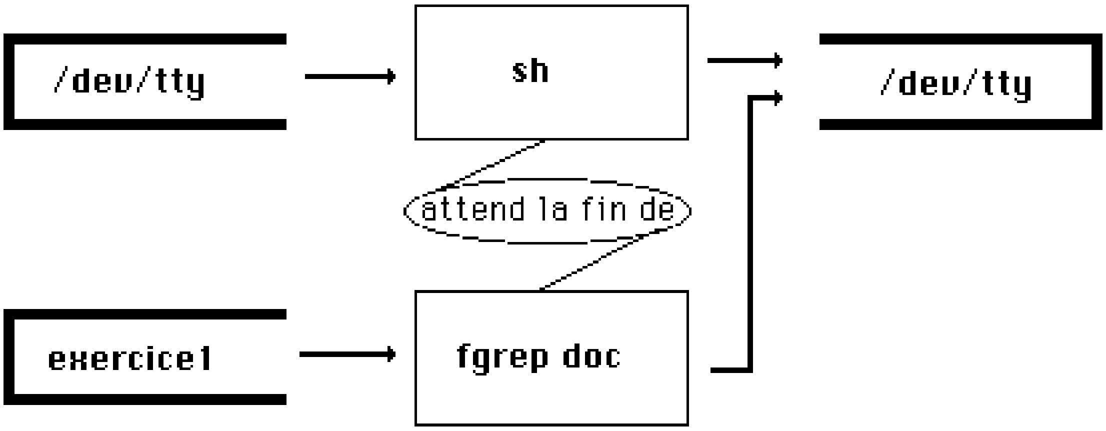


La tâche de listage crée un flux de caractères exploité par la tâche de filtrage (**fgrep doc** ne transmet en sortie que les lignes contenant la séquence contigue : de lettres d, o et c). Comme précédemment, l'interprète de commandes (**sh**) attend la fin de ses deux tâches filles avant de requérir une nouvelle commande ; par contre le filtrage ne vivra que tant que le listage continuera de l'alimenter en lignes à scruter.

La cascade ne permet que de créer des chaines linéaires de tâches et est spécifiée syntaxiquement, pour /**bin**/**sh**, par une barre verticale | séparant les tâches connectées. Un autre mécanisme dit de détournement (ou redirection) des flux permet que les tâches engendrées ne soient plus liées, en entrée comme en sortie, au fichier spécial /**dev**/**tty**. Ainsi la commande :

```
fgrep doc <exercice1
```

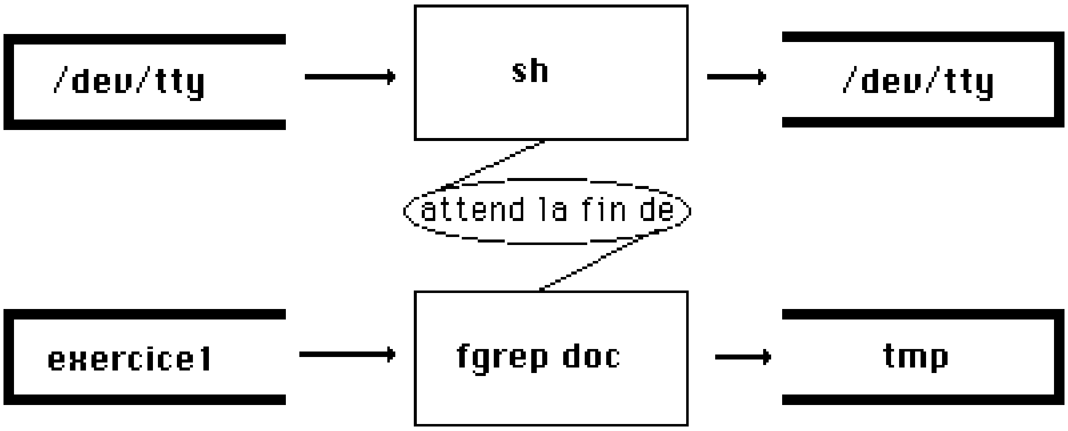
crée-t'elle :



Il y a eu détournement du flux d'entrée de la tâche de filtrage grâce à la spécification <**exercice1** apparaissant dans la commande. Les sorties, comme tout autre flux, peuvent aussi être détournées comme le montre :

```
fgrep  doc  >tmp  <exercice1
```

qui engendrera :



Les sorties engendrées par **fgrep doc** formeront le fichier de nom **tmp** (l'acronyme **tmp** est, dans le monde UNIX, l'abréviation usuelle pour temporaire.)

Qu'ils soient entrants ou sortants, une tâche peut disposer d'au plus une vingtaine de flux numérotés à partir de zéro. Les trois premiers sont usuellement nommés :

- **stdin** (pour *standard input*) et porte le numéro zéro,
- **stdout** (pour *standard output*) et porte le numéro un,
- **stderr** (pour *standard error* output) et porte le numéro deux.

Les sorties normales sont usuellement faites sur **stdout**, les comptes-rendus d'erreur sur **stderr**. Symétriquement les lectures sont effectuées sur **stdin**. Tous les flux peuvent être redirigés par une spécification. Dans ce qui suit les crochets carrés [et] signalent une éventualité ; les alinéas { et } une alternative. Ainsi *numéro de flux* est optionnel tandis que le symbole désignant le sens du flux est obligatoirement > ou <.

numéro de flux > *fichier*
numéro de flux < *fichier*

Ainsi

```
fgrep doc <exercice1 >tmp
```

est-il analogue à (l'ordre des spécifications de détournement est inimportant)

```
fgrep doc 1>tmp 0<exercice1 2>/dev/tty
```

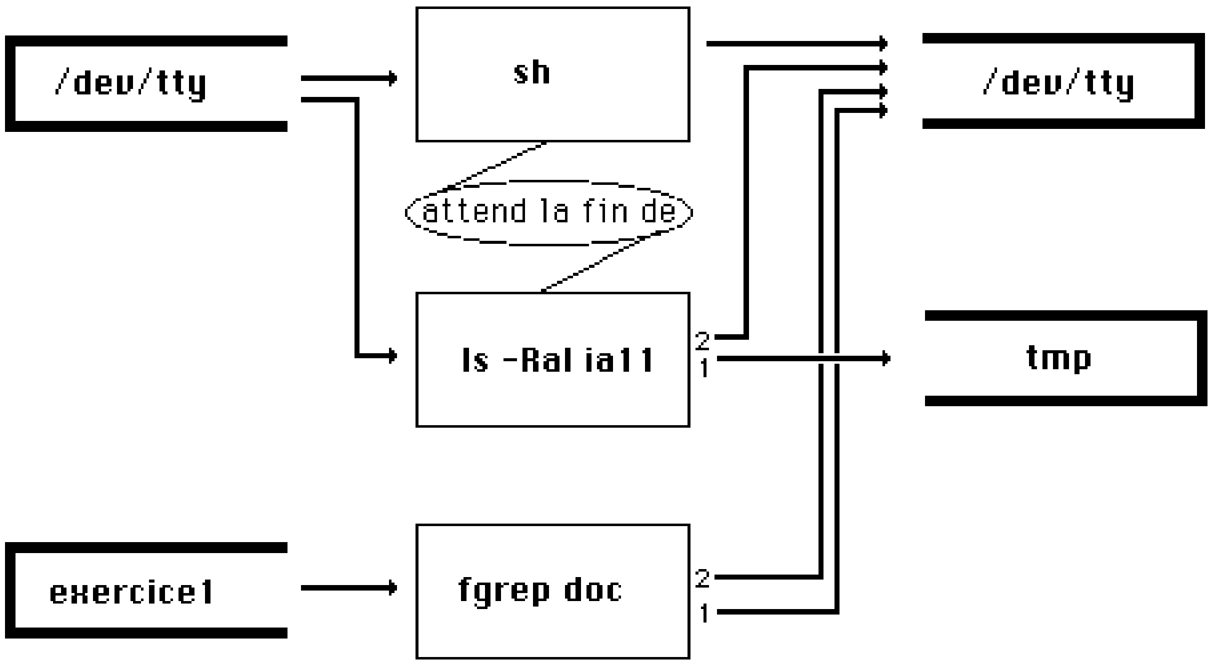
ce qui permet d'obtenir les messages d'erreurs sur l'écran et le résultat du travail normal de filtrage dans le fichier **tmp**.

L'opérateur | permet donc de connecter le flux **stdout** (1) d'une tâche au flux **stdin** (0) de la tâche suivante.

Dans tous les exemples précédents, la tâche mère était mise en attente de la fin des tâches filles. Le spécificateur **&** permet de supprimer cette contrainte et donc d'avoir plusieurs tâches simultanément actives. Ainsi

```
fgrep doc <exercice1 & ls -Ral ia11 >tmp
```

crée-t'il

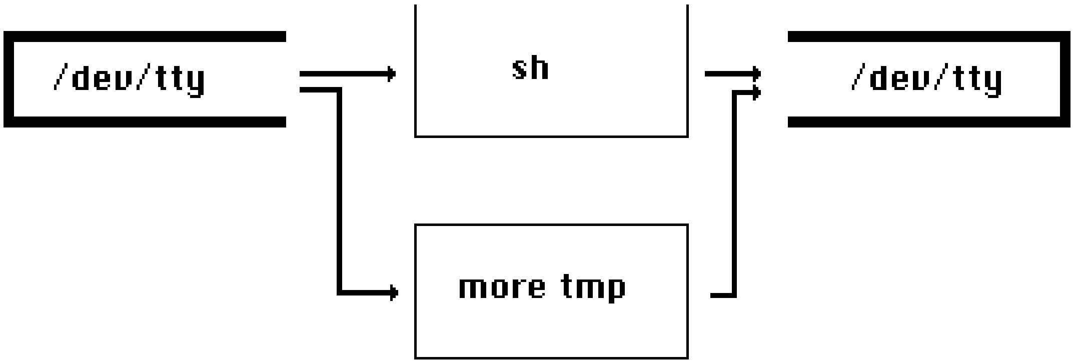


où les tâches de filtrage et de listage se dérouleront simultanément. L'écran étant partagé par ces deux seules tâches actives, les sorties du filtrage et les comptes-rendus d'erreur du listage seront affichés mélangés. Lorsque cette dernière sera achevée et indépendamment du filtrage qui pourra se poursuivre, l'interprète de commande se réveillera et sollicitera une nouvelle commande.

Le partage de **/dev/tty** peut conduire, si mal géré, à des situations dangereuses. Lorsque plusieurs tâches débitent sur l'écran, il peut être difficile de reconnaitre la paternité des diverses sorties. Un écran intéressant peut disparaitre, « poussé » par un texte routinier ! Mais lorsque plusieurs tâches se concurrencent pour lire le clavier, la situation peut devenir bien pire car on ne peut assurer vers quelle tâche ira tout caractère frappé ! C'est pourquoi, dans les exemples ci-dessus, à tout instant, il n'y avait qu'une seule tâche au plus pouvant requérir une entrée au clavier : **sh** en sommeil ne peut lire, **ls** et **fgrep** ne requièrent rien en entrée. L'exemple suivant exhibe une compétition entre deux tâches.

```
more tmp &
```

qui engendre :



Diverses manières existent de s'en sortir suivant la nature des tâches en compétition, par exemple l'appui répété sur la touche 'DEL' ou 'ANNULATION' (cf. signalisation). Les cas les plus graves nécessitent de tuer toutes les tâches d'un utilisateur depuis une autre console !

Divers programmes permettent de mieux manipuler flux, fichiers et commandes. On peut notamment convertir entre eux ces mêmes flux, fichiers ou commandes. Le tableau suivant recense ces utilitaires et les principales variantes :

pour convertir un ↓ en un →			
flux		> tee	sh ‘
fichier	< cat	cp mv	sh
commande		echo	eval
	flux	fichier	commande

Ainsi la duplication d'un fichier peut-elle s'effectuer de multiples manières.

–
```
tee <source >destination
```
commande sans argument qui use de deux flux.

–
```
cat source >destination
```
commande dont on détourne la sortie.

–
```
cp sortie destination
```
la commande spécialisée pour ce faire.

2.4 UNIX, le grand muet

Certaines revues américaines considèrent UNIX comme non convivial [*J'écrivais cela en 1985 !*] Le premier reproche qui lui est fait est de ne pas être verbeux. En effet un appel à un compilateur ne produit rien : ni bannière annonçant des choses aussi futiles que la date ou la version du compilateur, ni compte-rendu d'absence d'anomalies. Ces renseignements avaient naguère leur intérêt lorsque l'on travaillait par listages et traitement par lots (*batch*). La raison pour laquelle UNIX est lapidaire est claire. Une cascade de deux tâches nécessite que seules les informations importantes transitent dans le flux qui les lie. Il serait par trop compliqué d'éliminer les bannières et autres verbosités superflues afin d'extraire ce qui importe. De plus cela nécessiterait que les filtres sachent qui a engendré le flux afin de savoir où se trouvent les informations. La solution dite orthogonale (tout outil peut être abouché à tout autre outil) consiste à ce que les seules informations pertinentes transitent sur les flux appropriés (**stdin** et **stdout**). Il est donc normal que toute commande qui se déroule normalement ne produise rien (mais retourne le code zéro, cf. **exit**).

Cette notion de cascade implique celle de filtre. Un filtre est un programme qui exploite un unique flux entrant pour alimenter son unique flux sortant. Il ne vit que la durée de son flux entrant.

Le second reproche est qu'UNIX manque de gadgets tels que souris, menus déroulants, fenêtres en tout genre ... UNIX est né au même moment que ces innovations (Alan KAY et le Dynabook), auxquelles il a fallu dix ans aussi pour s'imposer. Rien n'empêche qu'un interprète de commandes fasse usage de ces techniques et certains le font (Aegis d'Appolo par exemple). Toutefois si elles simplifient l'emploi, elles ne se prêtent que fort mal à l'algorithmisation d'un tour de main, qualité qu'UNIX a au plus haut point poussée. Disposer d'un grand nombre d'utilitaires orthogonaux (qui ne font qu'une chose mais bien) et pouvoir en programmer tout enchainement répétitif ou conditionnel sur des objets dynamiquement paramètrables : voilà une force essentielle d'UNIX.

2.5 Diverses commandes

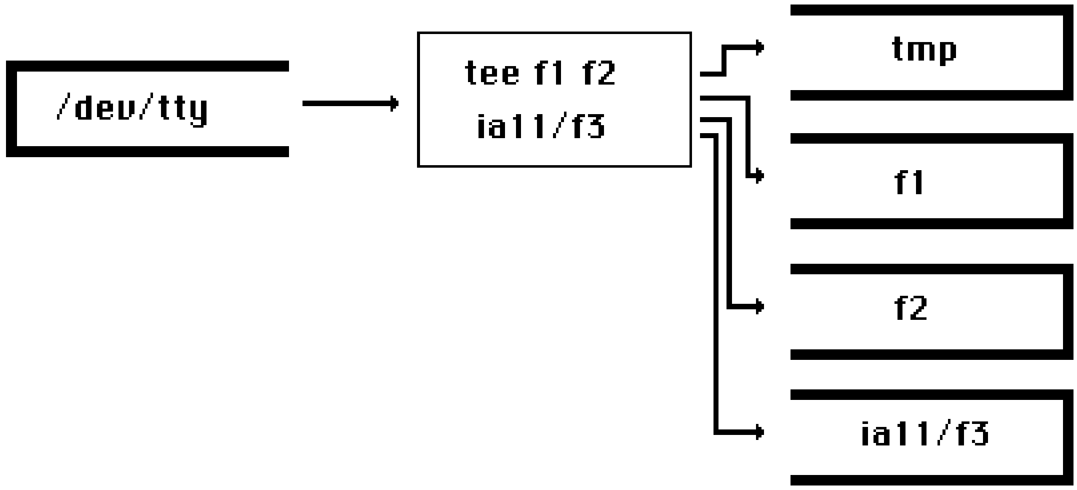
Les seules commandes qui seront ici survolées sont celles du tableau précédent, essentielles pour manipuler flux et fichiers.

2.5.1 tee

tee, qui est la prononciation anglaise de la lettre T, prend en arguments des noms de fichiers et recopie, sans altération aucune, son flux d'entrée dans chacun de ces fichiers ainsi que sur son flux de sortie normal. En français ce T correspond plutôt à un Y (ou patte d'oie) puisqu'à un unique flux entrant sont associés de multiples flux sortants. L'appel :

```
tee f1 f2 ia11/f3 >tmp
```

correspond au schéma



tee sans argument est le filtre identité ! **tee** est utile pour mémoriser des résultats intermédiaires lorsque, dans une cascade, un même flux doit être exploité de deux façons différentes. Par exemple :

```
... | tee avant | filtre | tee après | ...
```

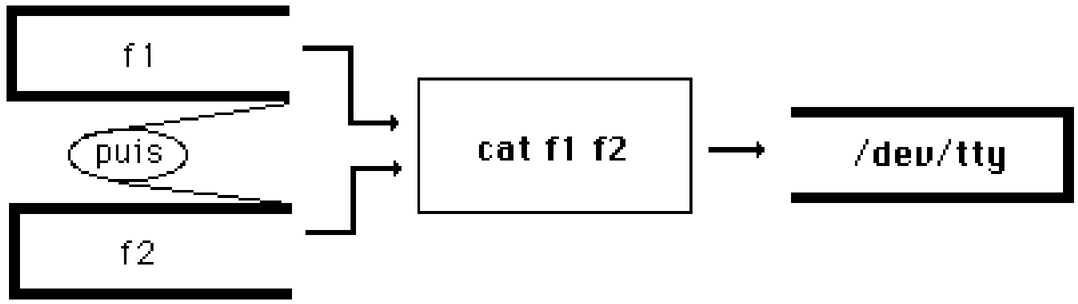
On pourra comparer les fichiers de trace (**avant** et **après**) par **cmp** (avec **cmp** pour CoMParer.) ou **diff** (avec **diff** pour DIFFérer.)

2.5.2 cat

cat (pour *catenate*c'est-à-dire concaténer) place la suite des contenus des fichiers spécifiés en arguments dans son flux de sortie. Ainsi :

```
cat f1 f2
```

conduit-il à



L'utilitaire **cat** met à profit les primitives d'ouverture, lecture et fermeture dynamique de fichiers offerts par le cœur d'UNIX. **cat** ignore son flux entrant. **cat fichier** | est analogue à la spécification <**fichier** mais peut s'étendre au cas de plusieurs fichiers.

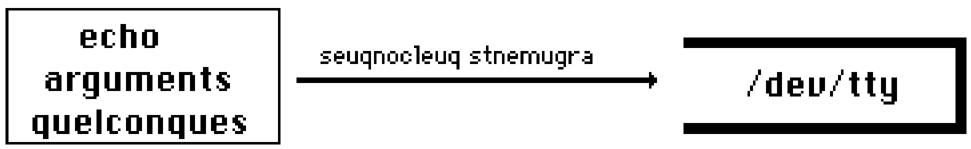
cat sert souvent à l'envoi direct de fichiers sur imprimante (à déconseiller lorsque plusieurs utilisateurs travaillent ensemble, dans ce cas préférer **lpr**, cf. **/dev/lp**). En revanche **cat** peut servir à lister des fichiers courts (tenant sur l'écran) sinon préférer **more** ou même un éditeur de textes.

2.5.3 echo

Comme son nom l'indique, **echo** fait écho de ses arguments. Ainsi :

```
echo arguments quelconques
```

correspond-t'il au schéma :



echo est pratique pour visualiser le résultat d'une expansion (cf. expansion) ou pour poser une question :

```
echo -n "La bourse ou la vie ? "
read reponse reste
```

echo n'exploite pas son flux entrant.

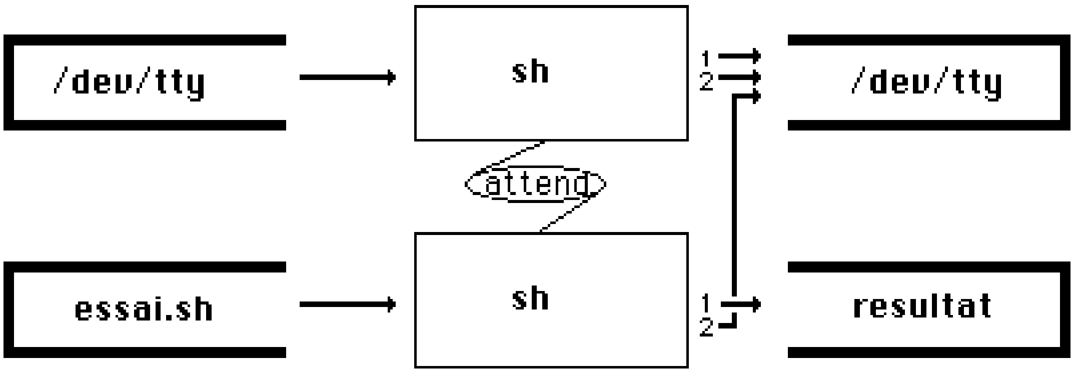
2.6 L'interprète de commandes sh

2.6.1 Fichiers de commandes

sh est l'interprète de commandes. Il peut se mettre en œuvre lui-même. Ainsi

```
sh <essai.sh >resultat
```

correspond au schéma global



Les commandes présentes dans le fichier **essai.sh** (l'extension **.sh** signifie que le fichier contient des commandes destinées à **sh**) seront exécutées et leur résultat sera placé dans le fichier **resultat**.

Une autre forme existe pour exécuter un fichier de commandes :

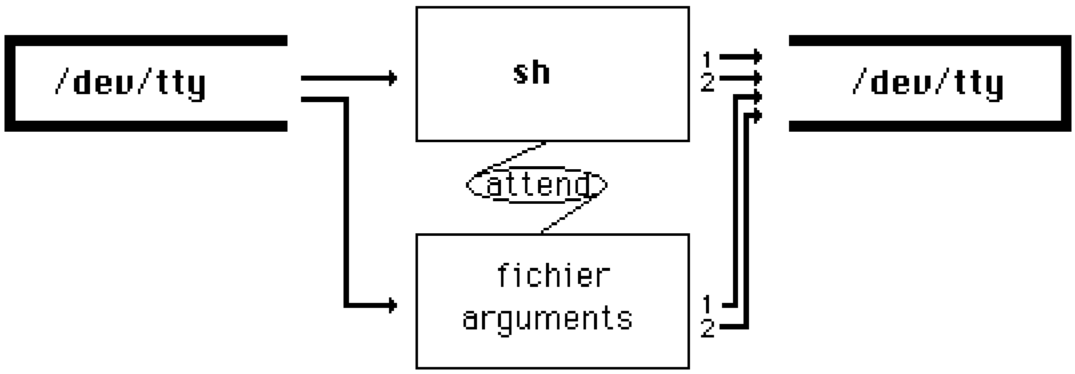
```
sh fichier
```

La différence réside dans les valeurs de la commande intrinsèque **read** qui lit une ligne dans le flux entrant (**essai.sh** dans le premier cas et **/dev/tty** dans le second).

Il est tellement usuel de posséder des fichiers de commandes que l'on peut en demander l'exécution en nommant seulement le fichier à condition de l'avoir rendu exécutable (cf. **chmod**). Ainsi

```
fichier arguments
```

conduit-il à



De fait puisqu'un programme exécutable est aussi archivé comme un fichier, l'interprète de commandes lorsqu'il cherche à évaluer une commande comme

```
fichier arguments...
```

regarde tout d'abord si le fichier possède l'attribut exécutable puis s'il correspond à des instructions binaires ou à un texte. Suivant le cas il lance l'exécution du programme ou bien celle d'une tâche basée sur **sh** lisant ce fichier considéré comme une suite de commandes. Cette tâche se terminera lorsque le fichier sera épuisé.

L'interprète de commandes (**sh**) offre des primitives de programmation structurée :

– La séquence (dénotée par **;**)

  ```
  ls -Ral ; echo fini
  ```

– Le bloc (débutant par **et s'achevant par**)

  ```
  { ls -Ral ; echo fini ; }
  ```

  C'est une séquence, considérée comme un objet unique qu'une seule spécification de détournement suffit à qualifier (cf. exemple tailles et greffes).

– La conditionnelle classique (if-then-else)

```
if f77 essai.f
then echo fini
else echo erreur
fi
```

La fin de ligne est considérée comme une fin de commandes ainsi que le point-virgule. On pourrait aussi écrire comme suit les exemples précédents :

```
if f77 essai.f ; then echo fini ; else echo erreur ; fi
```

et

```
{ ls -Ral
  echo fini
          }
```

La conditionnelle, ainsi que les primitives **while** et **until**, ne se peuvent concevoir que parce que chaque tâche qui s'achève retourne à son initiatrice un code :
– zéro, si tout s'est bien passé,
– une valeur non nulle, dans tous les autres cas.
Attention, sous **sh** les valeurs de vérité sont représentées à l'inverse des conventions usuelles. Pour C, Vrai est 1 et Faux, 0. Pour Pascal, Faux est avant Vrai et usuellement représenté aussi par 0.
La commande **exit** permet d'achever une suite de commandes en imposant le code de retour. Par défaut ce code est nul. Par exemple

```
if f77 essai.f
then exit 1
     echo fini
else
     echo "c'est bien erroné"
     exit   # comme exit 0 !
fi
```

Le retour sera Faux si la compilation s'est bien passé (et le message **fini** ne sera pas produit). Dans le cas contraire c'est le message **c'est bien erroné** qui sera engendré et la valeur logique de retour sera Vrai !

2.6.2 Autres possibilités

De nombreuses autres possibilités existent, qui font de l'interprète de commandes un véritable interprète de langage : Le langage *shell* ! La notion de variable existe. Ainsi :

```
prg=/u/untel/f77
```

déclare une variable nommée **prg** ayant pour valeur la chaine de caractères **/u/untel/f77**. On peut utiliser cette variable en la préfixant du symbole **$**.

```
f77 $prg/e.f
```

est équivalent à

```
f77 /u/untel/f77/e.f
```

C'est une simple substitution textuelle qui ignore la structure même des commandes. Ainsi

```
$ cde1='cat /'
$ cde2='o "Bonjour"'
$ ech$cde2
Bonjour
$ ${cde1}u/untel/f77/test.$cde2
```

listera le fichier **/u/untel/f77/test.o** et le fichier **Bonjour**.
Certaines variables prédéfinies existent comme **$0**,**$1**, **$2**...**$9**, **$PATH**, **$USER** ... Les dix premières permettent de nommer l'argument de rang correspondant lors de l'appel à une commande. Ainsi peut-on

faire des « sous-programmes » pouvant prendre facilement jusqu'à neuf arguments (**$0** est le nom du sous-programme appelé). On peut mettre en un fichier nommé **compil** le programme (qu'UNIX nomme *script* !) suivant :

```
if f77 $1
then exit
else { print $1 & echo $1 est erronné
                  exit 1
                  }
fi
```

puis appeler ce fichier par la commande

```
compil e.f
```

$1 sera connue comme étant la chaine de caractères **e.f** durant tout le déroulement de la commande **compil**. **$0** sera **compil**, les autres **$***i* seront vides. **compil** compilera **e.f** et en cas d'erreur

- lancera, en parallèle, l'impression du fichier erroné sur /**dev**/**lp**,
- produira un message explicatif,
- et, enfin, retournera le code d'erreur 1 (Faux) qui pourra être employé ultérieurement.

$PATH est l'ensemble des chemins d'accès où chercher les fichiers exécutables pour un utilisateur (qui a son nom contenu dans la variable **$USER**). Par exemple, lors de la soumission d'une commande

```
fichier arguments
```

sh recherchera **fichier** dans tous les répertoires mentionnés dans **$PATH** et ceux-là seulement. Si **$PATH** est vide, vous ne pourrez vraisemblablement pas faire grand chose ! Usuellement **$PATH** contient

```
:/usr/local:/bin:/usr/bin:/usr/ucb
```

Les répertoires sont séparés par **:**. Un répertoire vide désigne le répertoire courant. Ici, une commande mentionnant **fichier** est recherchée

- dans le répertoire courant,
- à défaut, dans le répertoire /**usr**/**local**, puis /**bin**, etc ...

On peut bien sûr étendre **$PATH** en écrivant par exemple

```
PATH=$PATH:/u/queinnec/cmds
```

(**cmds** est le répertoire où je regroupe mes commandes).

$MAIL contient le nom du fichier boîte à lettres que l'interprète scrutera toutes les cinq minutes afin de vous prévenir de l'éventuelle arrivée de messages.

Attention, une variable est locale à la tâche où elle a été définie. Pour qu'elle soit connue des tâches filles qui viendraient à être créées, elle doit être exportée par la commande

`export` *variable, variable ...*

En aucun cas vous ne pourrez modifier la valeur d'une variable d'une tâche ancêtre.

Par exemple, pour que la modification de **PATH** précédement faite soit connue de toute fille qui viendrait à être engendrée, il faudra exporter cette variable, et pour cela écrire :

```
export PATH
```

Un autre mécanisme bien utile est que le **sh** de connexion exécute systématiquement, s'il le trouve dans le répertoire initial (valeur de **HOME**),le fichier de commandes **.profile**. **.profile** peut donc contenir toutes les initialisations personnelles que vous pourriez souhaiter. À titre d'exemple, voici mon **.profile** personnel [*Celui de 1985 !*]

```
#
#       variables diverses
#
umask 022               # chmod go=r u=rw
PATH=":$HOME/cmds:/usr/local:/bin:/usr/bin:/usr/ucb:/etc"
#                        diverses options
```

```
SITE='Ensta'    # nom du site
SPEED=300       # Bauds
ED=emin         # Le préféré sur ce site
MORE='-cf'
#                       Les invites
if test $TERM = FT
then    PS1='

0315

031'<' # dessine un MU suivi d'un OMEGA
else    PS1="$SITE<- "
fi
PS2='Suite <- '
#                       message de bienvenue
echo "B I E N V E N U E sur $SITE"
#
trap ". $HOME/.eliforp" 0     # .logout en quelque sorte !
#
stty lnext "undef" nxpage 'undef' nxhalf 'u'
stty nxline 'u' flush 'u' erase '^H' kill '^U'
stty werase '^W' rprnt '^R' intr '^?' quit '^

'
stty eof '^D' -pageen -pageon ctlecho
#                                courrier automatique
MAIL=/usr/spool/mail/$USER
if test -s $MAIL
then    echo "$PS1 mail"
        mail
fi
# les exportations
export MORE ED MAIL SITE SPEED TERM PATH PS1 PS2 USER SHELL
```

2.6.3 Expansion

Le mécanisme d'expansion est une des plus heureuses caractéristiques de **sh**. L'expansion permet de nommer simplement des groupes de fichiers. Elle se fonde principalement sur l'emploi de deux caractères : **?** et *****.

- **?** remplace n'importe quel caractère,
- ***** remplace n'importe quelle séquence de caractères.

Si le répertoire local (cf. le nom local) contient les fichiers :

```
a       ab      acb     aard    att     itt
t       t1      t2      t3      t6      t7
```

alors

- **?** équivaut aux fichiers dont le nom ne se compose que d'un unique caractère, soit **a** et **t**.
- **? ? ? ?** désigne les fichiers dont le nom comporte quatre lettres exactement, c'est donc le seul fichier **aard**.
- **t*** regroupe tous les fichiers dont le nom débute par la lettre **t**, c'est-à-dire **t**, **t1**, **t2**, **t3**, **t6**, **t7**.
- ***t** regroupe symétriquement les fichiers dont le nom se termine par la lettre **t**, c'est-à-dire : **t**, **att**, et **itt**.
- D'autres combinaisons sont encore possibles comme ***t*** qui représente les fichiers contenant la lettre **t**.

On peut facilement voir le résultat d'une expansion grâce à **echo**

```
echo *t*
```

provoque

```
att     itt     t       t1      t2      t3      t6      t7
```

D'autres expanseurs plus spécifiques existent comme

```
t*[2-5]
```

qui sélectionne les fichiers dont le nom commence par **t** et qui se terminent par un chiffre compris entre **2** et **5**. Soient pour l'exemple précédent, **t2** et **t3** Cette expansion est faite de façon intelligente suivant les répertoires concernés. Ainsi

```
echo ../pascal/t*
```

expansera la spécification **t*** dans le répertoire **pascal**, frère du répertoire courant.

2.6.4 Déroulement

Lors de l'analyse d'une commande, la substitution de variables est effectuée en premier puis viennent les expansions mais, attention, le premier terme de la commande ne doit pas subir d'expansion. Ce premier mot est soit une commande intrinsèque soit un nom de fichier, Les mots suivants, c'est-à-dire des séquences de lettres séparées par des blancs ou des tabulations ou sauts à la ligne (ou, plus exactement, les caractères figurant dans la valeur de la variable **IFS**) sont alors ses arguments. Les répertoires figurant dans **PATH** sont alors scrutés séquentiellement afin d'y rechercher un fichier dont le nom est celui du premier mot et qui sera, si trouvé, lancé.

Tout ce qui précède n'est qu'une présentation succincte de l'interprète de commandes **sh**. D'autres existent, plus élaborés, qui ont pour nom **csh**, **nsh**, **vsh**, **tcsh**, **cwsh**...Tout interprète a ses détracteurs comme ses prosélytes ! Mais dans tous les cas ce ne sont que des programmes au dessus du cœur d'UNIX et tous peuvent s'exécuter en parallèle pour le bonheur de chacun. La variable **SHELL** vous renseignera sur votre interprète de commandes courant.

Vous êtes vivement invité à méditer les quelques sept pages de la documentation standard UNIX décrivant **sh**. ainsi qu'à étudier les fichiers de commandes qui sont accessibles (par exemple sous /**usr**/**local**).

2.7 Programmation de commandes

La programmation pour **sh** est semblable à toute programmation. Chaque problème nécessite une analyse puis l'élaboration d'une méthode de résolution et enfin l'adaptation d'icelle aux moyens que procure le langage cible. Considérant que le langage de **sh** est suffisament proche d'autres langages courants, nous ne donnerons que trois exemples commentés illustrant diverses techniques classiques et quelques autres possibilités non mentionnées de **sh**.

2.7.1 Du beau courrier

mail est un utilitaire permettant de lire ou écrire des messages. Pour lire son courrier il suffit de commander

```
mail
```

mail inspecte la boîte à lettre de l'usager (qui est en général le fichier /**usr**/**spool**/**mail**/**$USER**) et permet de consulter, détruire, sauvegarder les messages qui s'y trouvent (cf. **.profile**).

Pour envoyer un message, la commande sera

`mail` *destinataire1 destinataire2 ...*

mail se comporte comme un filtre c'est-à-dire qu'il considère son flux d'entrée comme la lettre à expédier. Il le lira donc jusqu'à déceler une fin de fichier (cf. ↑ **D**). Ce comportement, s'il est simple, n'est cependant guère optimal lorsqu'on désire écrire une belle lettre. On use alors de

`mail` *destinataire1 destinataire2 ...* `<lettre}`

où **lettre** sera un fichier que l'on aura composé, par exemple, sous éditeur de textes.
Voici le programme

```
#
# Usage : mail destinataire ...
#
MAILER=/bin/mail                # le véritable mail !!!
if  test  $#  =  0
then    # S'il n'y a pas d'argument, c'est une simple
        # lecture du courrier
        $MAILER
else    # Préparer la forme de la belle lettre
        cat  $HOME/.mailproto  >/tmp/mail$$  2>/dev/null
        # L'écrire
        ${ED-/bin/ed}  /tmp/mail$$
        # L'envoyer avec copie pour soi
        $MAILER  $USER  $*  </tmp/mail$$
        # Nettoyer son passage
        rm  /tmp/mail$$
fi
```

Les remarques qui suivent éclairent ce programme.

- **MAILER** est une variable de **sh** qui contient le nom du véritable utilitaire de courrier. Il est nécessaire que ce soit le nom complet afin qu'il n'y ait nulle ambiguïté sur lui. En effet mettre **mail** tout seul renverrait à la commande même que nous explicitons et ferait ainsi boucler l'interprète, puisque **mail** serait recherché avec les mêmes règles qui l'ont fait trouver ! L'intérêt est que cette nouvelle commande pourra se nommer comme l'ancienne qu'elle exploite : **mail** !
- **$#** est une variable dynamique prédéfinie de **sh**. Sa valeur est une chaine de caractères représentant (en base décimale) le nombre d'arguments d'appel à la commande courante. Deux cas sont à distinguer suivant que **mail** est invoqué sans argument ou avec des destinataires.
- **test** est un programme particulier dont le rôle est de rendre un code de retour égal à zéro ou un, c'est-à-dire (respectivement) Vrai ou Faux . **test** détermine sa valeur après analyse de ses arguments d'appel. Ceux-ci sont ici :

  une chaine de caractères obtenue par valeur de **$#**,
  la chaine de caractère formée du seul signe =,
  la chaine de caractères formée du caractère **0**.

  Ces trois arguments spécifient à **test** d'effectuer une comparaison dont le résultat sera exploité par **if**.
- On utilise, pour préparer sa lettre, un fichier temporaire que l'on place tout naturellement dans le répertoire **/tmp**. **$$** est une autre variable prédéfinie de **sh** dont la valeur est la chaine de caractères représentant le numéro de la tâche qui exécute **mail**. On garantit ainsi que deux utilisateurs différents utilisant ensemble **mail** ne se gênent point puisqu'ils travailleront l'un comme l'autre avec des fichiers temporaires différents.
- Le fichier temporaire est créé par recopie d'un fichier particulier de l'utilisateur nommé **.mailproto** et situé dans son répertoire origine (La plupart des programmes qui ont besoin de données valables quelque soit l'endroit d'où ils sont appelés, placent ces données dans des fichiers invisibles (dont le nom débute par un point) situés sous le répertoire origine. Par exemple, **emacs**, **msgs**, **winnie**, **sh** ...) (**$HOME**) Le flux d'erreur est détourné sur **/dev/null** afin que si erreur il y a (la plus probable étant que **.mailproto** n'existe pas) elle soit invisible.
- La lettre contient alors un fond (par exemple, une entête, un colophon ...) modifiable par un éditeur de texte. Celui-ci est déterminé par la valeur de la variable **ED** qui contient le nom de votre éditeur préféré. **${ED}** est équivalent à **$ED** mais **${ED-/bin/ed}** permet de spécifier que si la variable **ED** n'existe pas, ce sera l'éditeur standard d'UNIX qui sera utilisé.
- **$*** est une autre variable dynamique prédéfinie de **sh**. Elle équivaut à la suite des arguments **$1 $2 ...$9**. Elle représente donc ici la liste des destinataires de la lettre.

– Enfin lorsque vous sortirez de l'editeur, votre lettre sera expédiée et le fichier temporaire effacé.

2.7.2 Fouiner

Le programme qui suit permet d'extraire tous les fichiers lisibles accessibles depuis un répertoire. Il est formé d'un longue théorie de cascades.

```
#
# Usage : fouiner répertoire
#
# Produire tous les noms accessibles depuis le répertoire argument
find $1 -print  |
# ... puis pour chacun ...
while read f
   do
      file $f   # déterminer son type
   done         |
# n'en retenir que ceux qui sont des textes
sed -n -e '/ text/s!:   .*$!!p' |
# et pour chacun s'enquérir de sa taille
while read f
   do
      wc -c $f
   done         |
# Trier par ordre croissant de taille
sort - +0n +1d  |
# Eliminer les blancs superflus ...
sed -e 's!^ *!!'        |
# puis regrouper sur une seule ligne les fichiers de même taille ...
if read n f
then while read p g
        do
           if test $n = $p
           then f="$f $g"
           # préfixé de la commande "unique"
           else echo unique $f
                f="$g"
                n=$p
           fi
        done
        echo unique $f
fi      |
# puis exécuter
sh
```

fouiner use du programme **unique** qui suit :

```
#
# Usage : unique fichiers...
#
# Pour tous les fichiers arguments
for fichier
do
   # echo On compare le fichier $fichier
   shift
   for f in $*
```

```
    do
        # echo avec $f
        if cmp -s $fichier $f
        # On oublie fichier si semblable à l'autre
        then exec unique $*
        fi
    done
    # mais on le sort s'il est différent de tous les autres.
    echo $fichier
done
```

Encore quelques remarques.

- La première cascade pourrait, logiquement et plus simplement, s'écrire :

  ```
  file `find $1 -print`
  ```

  en usant du métacaractère accent grave (*backquote*) qui convertit un flux (le résultat d'une commande) en une liste d'arguments. Le malheur est que **sh** ne peut manipuler un trop grand nombre d'arguments (surtout s'il dépasse la centaine). Il faut donc les traiter par lots et le plus simple est de les considérer un par un.
- **find** est un utilitaire qui permet de rechercher un nom dans une arborescence. On peut associer à cette recherche des conditions sur la forme du nom, son type (fichier, répertoire ...), sa dernière date de modification, etc ...Ici comme aucune condition ne restreint le parcours, tous les noms sont imprimés sur des lignes différentes.
- La phrase :

  ```
  while read f
     do
        file $f
     done
  ```

  permet de lire cet ensemble de lignes et d'appliquer l'utilitaire **file** à chacun des fichiers mentionnés. **read f** affecte à la variable **f** le nom lu qui est alors soumis en argument à **file**.
- **file** est un autre utilitaire qui inspecte le début d'un fichier et tâche d'en déduire son type. Il reconnait les répertoires, les programmes exécutables et tente de déterminer le langage dans lequel sont écrits les fichiers contenant des textes. Le mot **text** apparait alors si le fichier semble lisible. Par exemple,

  ```
  $ file *
  cnv:    command text
  cupple: english text
  cupple.c:       C text
  cupq:   pure executable not stripped
  date:   pure executable stripped
  db:     tbl, eqn or roff text
  dir:    directory
  ```

  Les lignes produites contiennnent le nom du fichier suivie de **:** et d'une tabulation (↑ **I**) puis d'un ensemble de mots caractérisant la nature du fichier. On ne souhaite retenir que ceux de type texte aussi filtre-t'on le flux de sortie pour ne laisser passer que les lignes où figure le mot **text**. On supprime en outre tout ce qui suit le nom du fichier, désormais inutile.
- **sed** (pour *Stream EDitor*) est un filtre apparenté à l'éditeur **ed**, mais, à rebours d'un éditeur de texte classique où le fichier à traiter est connu et la suite des commandes à effectuer dessus inconnue, ici, la suite des commandes est connue. Chaque ligne du flux entrant est lue et successivement confrontée à toutes les commandes, modifiée en conséquence puis émise dans le flux sortant. Une seule commande permet ici d'effectuer le filtrage :

  ```
  / text/s!:     .*!!p
  ```

  ce qui se lit simplement ainsi :
- / **text**/ La commande qui suit ne s'applique qu'aux lignes contenant la suite des lettres « text » (le blanc qui précède autorise les noms de fichiers contenant les lettres « text »).

- **s** Substituer ...
- **:	.*** ...une séquence de lettres débutant par le caractère deux-points suivi d'une tabulation (peu visible ici) suivie d'une séquence de caractères quelconques (le point représente n'importe quoi, l'étoile déclare qu'on peut trouver l'objet précédent un nombre de fois quelconque. C'est la notation propre aux expressions régulières de Kleene)...
- **$** ...contrainte à se terminer en fin de ligne.
- **!!** La séquence de lettres, si trouvée, est remplacée par rien, c'est-à-dire effacée. **!** est ici le séparateur des deux parties de la substitution.
- **p** Enfin imprimer !
- Le flux ne comporte donc plus que des noms de fichiers lisibles. On pourrait s'arrêter là mais en fait de nombreux fichiers sont semblables (de même contenu). Ce qui suit permet d'éliminer ces doublons.
- Pour chacun de ces fichiers on extrait sa taille grâce à **wc** (pour *Word Count*) avec l'option **-c** car on ne désire que la seule taille en caractères.

```
$ wc -c *
   1256 u0
  17006 u1
   3483 u2b
```

- On trie le flux engendré grâce à **sort** (numériquement sur le premier champ et alphabétiquement sur le second) et enfin on regroupe ensemble tous les fichiers ayant même taille. Il est alors évident que deux fichiers de taille différentes ne peuvent être semblables aussi restreindra-t'on les comparaisons (coûteuses puisqu'effectuées octet par octet) qu'entre fichiers de tailles identiques.
- Le script de regroupement

```
if read n f
then while read p g
        do
           if test $n = $p
           then f="$f $g"
           # préfixé de la commande "unique"
           else echo unique $f
                f="$g"
                n=$p
           fi
        done
        echo unique $f
fi
```

s'il prend le flux

```
128 /u/qnc/bouquin/u0
128 /u/qnc/bouquin/save/u0
128 /etc/group
129 /u/scratch/#123
130 /usr/bin/.emacs_219
```

produira

```
unique /u/qnc/bouquin/u0 /u/qnc/bouquin/save/u0 /etc/group
unique /u/scratch/#123
unique /usr/bin/.emacs_219
```

- Enfin la commande **unique** permet d'éliminer les doublons pouvant figurer dans ses arguments. Il suffit d'exécuter le flux sortant pour que le tour soit joué !

Si vous désirez lire tout ce qui est lisible (et accessible, mais la plupart des choses le sont sous UNIX) il vous suffira d'ajouter au programme précédent la cascade suivante où chaque fichier pourra être affiché sur votre écran grâce à l'utilitaire **more**.

```
while read f
```

```
do
   more $f
done
```

more est un programme bien utile qui découpe un fichier en pages d'un nombre de lignes ajusté à l'écran que vous employez pour le lire. Sous **more** vous disposez de toute latitude pour lire posément, revenir en arrière ou chercher des chaines de caractères.

Bonne lecture !

2.7.3 Ce livre

Ce livre a été, comme il se doit, entièrement composé sous UNIX. La composition est un processus qui dispose des lettres, des phrases, des paragraphes au sein de pages. **nroff** est un formateur de textes effectuant ce travail avec comme cibles des imprimantes classiques. Un formateur de textes est un filtre qui traite des fichiers dans lesquels figurent des textes à composer et des directives de formatage associées. Ces directives indiquent les traitements à opérer sur ces textes (centrer, justifier à gauche et/ou à droite, mise en place de tabulations, de marges, numérotation de chapitres, élaboration de la table des matières, changement de polices ...). Les directives sont repérées par un point en tête de ligne. Il est possible sous **nroff** de créer ses propres directives par assemblage de directives prédéfinies. Des jeux tout faits et paramètrables existent donc comme **man** qui permet de formater le **man**uel de référence, ou encore **me** (dû à Eric.P. Allman de l'UCB) ici présentement mis à contribution : qu'il en soit remercié !

nroff peut lui-même être paramètré, au prix d'une recompilation, en fonction de l'imprimante visée pour cible. Tout comme les terminaux intelligents (cf. De l'intelligence des terminaux), les imprimantes reconnaissent diverses séquences de caractères ordonnant des actions comme des déplacements horizontaux ou verticaux, des changements de police ...Par exemple ↑ **L** fait sauter en tête de la page suivante. Cependant **nroff** ne peut être totalement paramètré pour la langue française en raison des signes diacritiques. Le programme **typo** a donc été créé afin de composer ce livre.

```
#
# Usage : typo  -imprimante fichiers ...
#
TYPO=/u/qnc/TYPO
type="$1"
shift
case  X$type  in
   X-IDS)
        cat  $TYPO/init.me  $TYPO/IDS.me  $*    |
        accent                                  |
        nroff -  -TIDS -e                       |
        $TYPO/IDS.me                  ;;
   X-FT)
        cat  $TYPO/init.me  $TYPO/FT.me  $*     |
        accent                                  |
        nroff -  -TFT  -e                       |
        $TYPO/FT.me                   ;;
   X-DY)
        cat  $TYPO/init.me  $TYPO/DY.me  $*     |
        accent                                  |
        nroff -  -TDY -e                        |
        $TYPO/DY.me                   ;;
   *)
        exit 1                        ;;
esac
```

typo est une cascade où les fichiers à traiter passent en premier lieu au travers du filtre **accent**. Le but de ce dernier est de manipuler correctement les signes diacritiques dont les accents. Ceux-ci sont, dans les fichiers à formater codés de la façon suivante :

é est écrit e' à est écrit a' c// est écrit c// œ est écrit o//e etc ...

Ainsi la phrase :

Un été sans noël haï en hâte et en œuvre

au sein de ce paragraphe s'écrirait :

```
.pp
Ainsi la phrase
.b
.ce
Un e'te' sans noe:l hai: en ha^te et en o//euvre
.r
au sein de ce paragraphe s'e'crirait :
```

.pp est une directive de **me** stipulant qu'un nouveau paragraphe débute. **.b** et **.r** basculent l'impression en police grasse (*bold*) ou normale (*roman*). **.ce** est une directive de nroff qui centre la ligne qui suit au milieu d'une ligne.

Le résultat de **accent** est de transformer tous ces codages en un codage approprié à **nroff**.

`e'` est transformé en `\o'e''`
`c//` est transformé en `\o'c,'`
`o//e` est transformé en `o\h'-0.2m'e`

Pour **nroff**, `\o` (pour *overstrike*) signifie qu'il lui faut empiler la frappe des caractères qui suivent : ainsi é est-il obtenu par frappe superposée de la lettre **e** et d'un accent aigu. `\h` (pour Horizontal) réalise un déplacement horizontal. Le « m » est une unité de distance typographique égale à la chasse de la lettre « m ».

Une fois ces transcriptions effectuées ainsi que diverses autres telles que l'insertion d'un demi-espace entre le dernier mot d'une phrase et le point final, etc ...on aboutit au formatage. Les appels à **nroff** sont paramètrés par le type de l'imprimante visée (FT signifie Fortune Terminal, IDS est l'imprimante PRISM 132, etc ...). Le fichier à formater est précédé du fichier standard **init.me** qui contient tous mes idiotismes généraux et du fichier **FT.me** ou **IDS.me** où se trouvent tous les paramètres dépendants de l'imprimante cible (taille du papier notamment).

Le résultat du formatage passe alors au crible d'un filtre dépendant de la cible, en effet divers travaux sont encore inachevés. Un <u>mot</u>souligné comme celui-ci est engendré par **nroff** comme la séquence m↑ **H**_o↑ **H**_t↑ **H**_. Chaque caractère est suivi d'un retour arrière puis souligné or, comme on ne peut généralement superposer des caractères sur un écran et si un attribut de visualisation « souligné » existe, il faut effectuer la conversion idoine. Par exemple, pour FT, le résultat sera ↑ \HPmot↑ \IP. Les deux séquences qui bordent le mot activent puis inactivent le mode souligné (cf. l'utilitaire **ul**). En ce qui concerne les polices de caractères, la police grasse sur DY (Olivetti DY311) est obtenue par ↑ \[E et supprimée par ↑ \[R mais aussi par une fin de ligne. Lorsqu'un paragraphe de plusieurs lignes est en gras, il est donc nécessaire de regénérer ↑ \[E en tête de chaque ligne ! Enfin en ce qui concerne les accents, la surimpression conduit à des séquences que l'on peut quelquefois remplacer par des lettres mieux dessinées. Sur DY, par exemple, existe sur la marguerite qui l'équipe un « à » beaucoup plus beau que ce que l'on obtient par a↑ **H**'.

Tous ces filtres sont des appels à **sed**.

Malheureusement toutes ces imprimantes, si elles fournissent des brouillons fort acceptables, n'atteignent pas une qualité typographique suffisante pour ce livre. Il faut pour cela utiliser des photocomposeuses ou imprimantes « laser » et pour cela abandonner **nroff** et passer à **troff**. (t pour *typesetter*). **troff** est compatible de **nroff** puisqu'il formate les fichiers de **nroff** avec les mêmes conventions.

La composition est un art difficile que ne rend pas simple la diversité des moyens de reproduction existants.

Les objets d'UNIX sont des fichiers, des flux et des tâches. On peut agencer, selon ses goûts et besoins, tous ces divers objets en schémas plus ou moins compliqués.

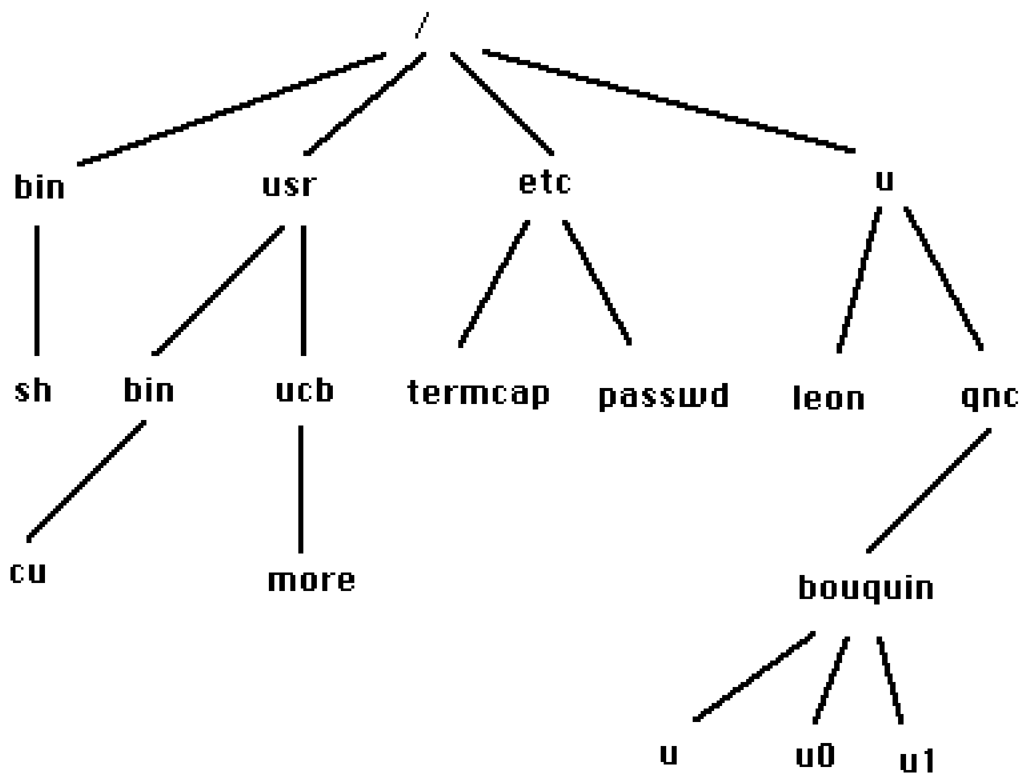
Un interprète de commandes n'est qu'un moyen, particulièrement efficace et simple, de mettre en œuvre ces schémas en les traduisant en une succession d'appels au noyau d'UNIX.

Chapitre 3

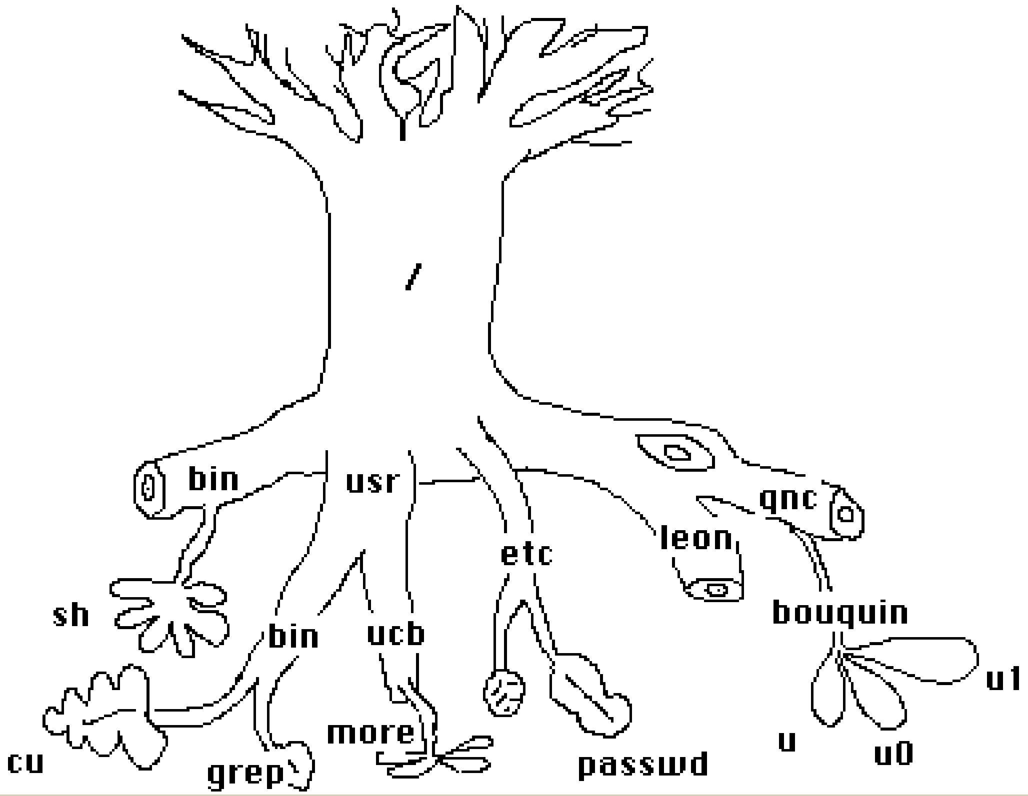
Système de fichiers

3.1 Introduction

Le système de gestion de fichiers d'UNIX est arborescent. Il se compose
– d'une racine (généralement dessinée en haut) et nommée, en anglais *root* quoique orthographiée /
– de branches ou répertoires ou encore catalogues (*directory*)
– de feuilles ou fichiers (*file*). Par exemple,



La même chose vue sous une forme véritablement arborescente :



les systèmes classiques associent, en général, un répertoire à chaque support physique, ce répertoire contient la liste de tous les fichiers présents sur le support. Sous UNIX, cette notion a été généralisée puisqu'un répertoire est un agrégat de fichiers ou de répertoires. Par exemple

/ **u bin usr ucb** sont des répertoires,
more sh termcap passwd sont des fichiers.

Un arbre est un exemple de structure hiérarchisée. L'intérêt en est de pouvoir regrouper ses fichiers par thèmes (un même langage, un même problème, un même utilisateur ...) et ainsi combattre l'inflation horizontale où un nombre important de fichiers coexistent sans rapport aucun. En outre certaines commandes permettent de facilement manipuler des groupes entiers de fichiers (cf. expansion) et donc d'aisément entretenir son arborescence personnelle. Par exemple

```
mv /u/queinnec /usr
```

taille la branche /**u**/**queinnec** et la greffe sur /**usr**.

Les fichiers normaux sont des suites d'octets éventuellement vides. Les répertoires sont des fichiers un peu particulier contenant les noms de leurs branches ou feuilles ainsi qu'un numéro (*inode*) permettant de les localiser sur le support physique. Les fichiers spéciaux servent d'interface avec l'extérieur et sont usuellement regroupés dans le répertoire /**dev** (comme *device*).

3.2 Le nom propre

Tout fichier possède un nom propre qui l'identifie au sein du répertoire où il figure. En effet, deux fichiers de même nom ne peuvent simultanément figurer dans un même répertoire. Sont des noms propres **bin, sh, ucb, etc** ...

3.3 Le nom complet

Tout fichier ou répertoire possède aussi un nom complet l'identifiant sans équivoque. Il s'obtient à partir du chemin que l'on doit suivre pour y accéder depuis la racine. Chaque nom propre de répertoire est suivi du séparateur /. Par exemple

```
/usr/ucb/more
/etc/passwd
//bin///sh
```

abondance de séparateurs ne nuit souvent pas !

3.4 Le nom local

Afin de simplifier la dénomination des fichiers, à toute tâche est associée un répertoire courant. Un fichier peut être nommé relativement à ce répertoire courant. Le nom local représente le chemin qui mène de ce répertoire à ce fichier.

Un fichier local au répertoire courant a un nom relatif confondu avec son nom propre. Par exemple, si le répertoire courant est /**u**/**queinnec** alors

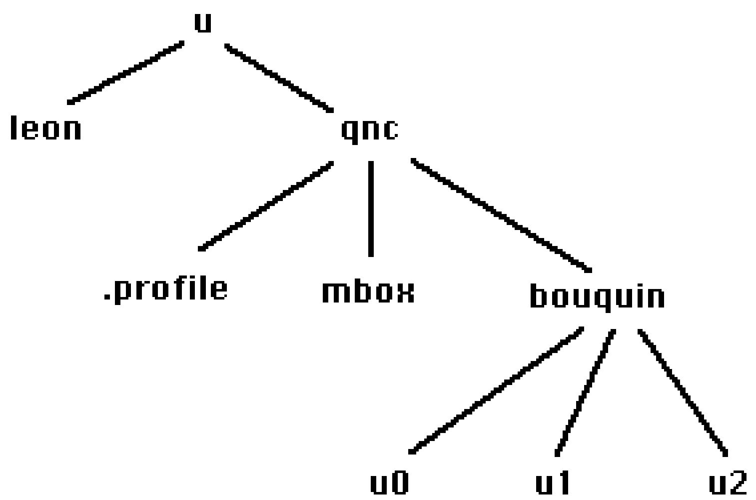
> **.profile** est le nom relatif ainsi que le nom propre de /**u**/**queinnec**/**.profile** **bouquin**/**u1** est le nom relatif de /**u**/**queinnec**/**bouquin**/**u1**.

Un nom local se distingue d'un nom absolu par son premier caractère qui n'est pas une barre de fraction !

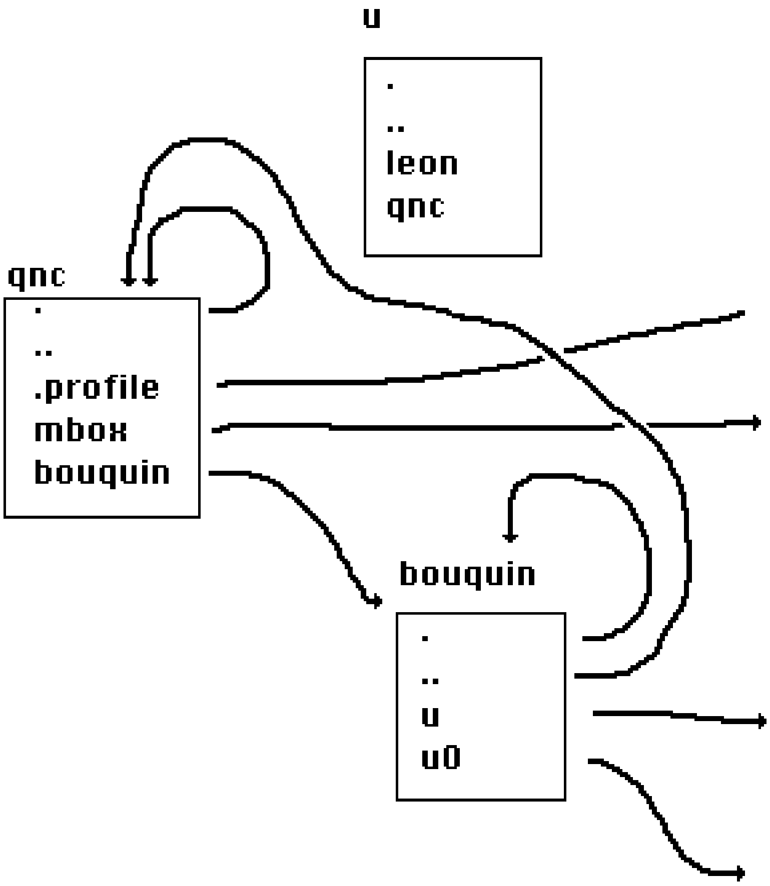
Comme il est nécessaire de pouvoir désigner les aïeux et les cousins du répertoire courant (aussi bien que lui-même), deux notations sont introduites.

- **.** désigne le répertoire courant. Ainsi **.**/**.**/**bouquin**/**u1** est un nom relatif pour /**u**/**queinnec**/**bouquin**/**u1**
- **..** désigne l'unique père du répertoire courant. Ainsi **..**/**leon**/**.profile** est le nom relatif de /**u**/**leon**/**.profile**.

Les notations peuvent se conjuguer **.**/**..**/**queinnec** est analogue à **.** tandis que l'énorme chemin qui suit **..**//**..**///**usr**/**..**/**u**/**queinnec**/**..**/**leon** s'énonce plus aisément comme /**u**/**leon** ! **.** et **..** sont deux entrées particulières de tout répertoire. La hiérarchie suivante



correspond à



Afin d'éviter que **.** et **..** ne soient systématiquement répondus à toute demande de listage comme **ls**, les noms de fichiers dont le nom propre débute par un point sont en général ignorés (notamment par l'expansion). La spécification ***** ne dénote que l'ensemble des fichiers du répertoire courant dont le nom propre ne débute pas par un point. En revanche ***** et **.*** les nomment tous sans exception et l'option **-a** du listage (**ls**) liste toutes les entrées d'un répertoire sans exceptions. Du coup **.profile** qui n'est usuellement pas un fichier de travail est, la plupart du temps, omis !

3.5 Les extensions

Le nom propre des fichiers se compose souvent d'un mot suivi d'une extension. Le type du fichier induit souvent l'extension.

- Un fichier écrit en *shell* peut posséder l'extension **.sh** (ce n'est pas toujours vrai car c'est alors trop long à frapper pour être le nom d'une commande).
- un fichier écrit en C possède l'extension **.c**,
- un fichier écrit en Pascal possède l'extension **.p**,
- un fichier écrit en FORTRAN possède l'extension **.f**,
- un fichier écrit en assembleur possède l'extension **.s**, (c'est cependant **.a** pour Pascal ETR)
- un fichier écrit en Le_Lisp possède l'extension **.ll**,
- Un fichier objet, issu d'un compilateur possède l'extension **.o**,
- etc ...

Un fichier d'initialisation pour un programme x se nomme souvent **.**x**rc**.

3.6 Navigation sylvestre

Suivant son intérêt du moment, l'on peut être amené à travailler dans telle ou telle partie de son arborescence. Un programme **pwd** (pour Print Working Directory) retourne le nom complet du répertoire courant. Il est possible de déplacer ce dernier grâce à la commande intrinsèque (**cd** ne peut être un programme car alors il ne changerait que le répertoire local de la tâche exécutant **cd** et non celui de la tâche mère **sh**). Il est donc nécessaire que **cd** soit directement interprétée par votre **sh** d'origine (cf. **exec**). **cd** (pour Change Directory). Par exemple,

```
$ pwd
/u/queinnec/bouquin
$ cd .. ; pwd   # déplacement relatif
/u/queinnec
$ cd /u/q*/b* ; pwd     # déplacement absolu (avec expansion)
/u/queinnec/bouquin
```

Puisque chaque tâche possède un répertoire courant, l'interprète de commandes en est doté d'un qui est le répertoire de travail de l'utilisateur. C'est ce dernier que déplace ou interroge **cd** ou **pwd**. À la connexion, la valeur initiale de ce répertoire est trouvée dans le fichier **/etc/passwd** (sixième champ) et devient la valeur de la variable **HOME**. **cd** employée sans argument est équivalente à **cd $HOME**.

Pour naviguer à vue, encore faut-il pouvoir inspecter son entourage. Le programme **ls** (pour LiSter) retourne la liste des entrées du répertoire courant. Ainsi une exploration s'effectue-t'elle par alternance de **ls** et **cd** jusqu'à trouver ce que l'on désire.

3.7 Tailles et greffes

On crée un répertoire grâce au programme **mkdir** (pour MaKe DIRectory). Par exemple, la commande

```
mkdir /u/nkw /u/nkw/own
```

crée sous **u** le répertoire **nkw**, et sous ce dernier **own**. Cet ordre de création est bien sûr à respecter ! La commande suivante crééra les mêmes répertoires :

```
"cd u ; mkdir nkw ; cd nkw ; mkdir own"
```

Lorsqu'un répertoire est créé, il est vide et comporte seulement les deux entrées **.** et **..**.

On supprime un répertoire vide grâce à **rmdir** (pour ReMove DIRectory). Par exemple on détruira les répertoires précédemment créés par

```
rmdir /u/nkw/own /u/nkw
```

dans cet ordre !

La suppression de fichier s'effectue similairement grâce à **rm** (pour ReMove).

La création de fichier ne passe pas par des commandes particulières. De nombreux programmes en créent (éditeurs de textes entre autres) ainsi que le détournement de flux (spécificateur $>$). La création de fichiers utilise le sous-programme **creat** du cœur d'UNIX. Si le fichier n'existait pas précédemment, il est créé sinon il est remis à vide.

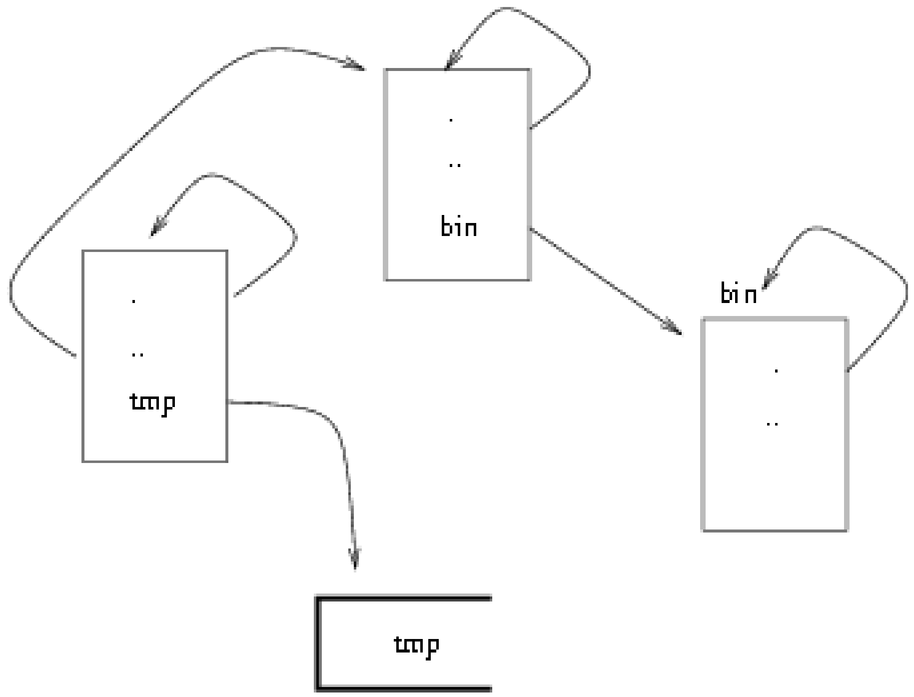
De nombreux programmes modifient les fichiers (éditeurs, metteur au point ...). Un important cas particulier est l'ajout incrémentiel. Le fichier se voit étendu par suffixation, sans modification aucune de la partie préfixe. Un spécificateur particulier » permet sous **sh** de détourner un flux en queue d'un fichier. Par exemple, les deux programmes suivants sont équivalents.

```
PROG=gros_calcul                # un programme très long
echo debut de $PROG     >log    # dont on veut enregistrer
date                    >>log   # la durée d'exécution
$PROG
echo fin de $PROG       >>log   # dans le fichier log.
date                    >>log
```

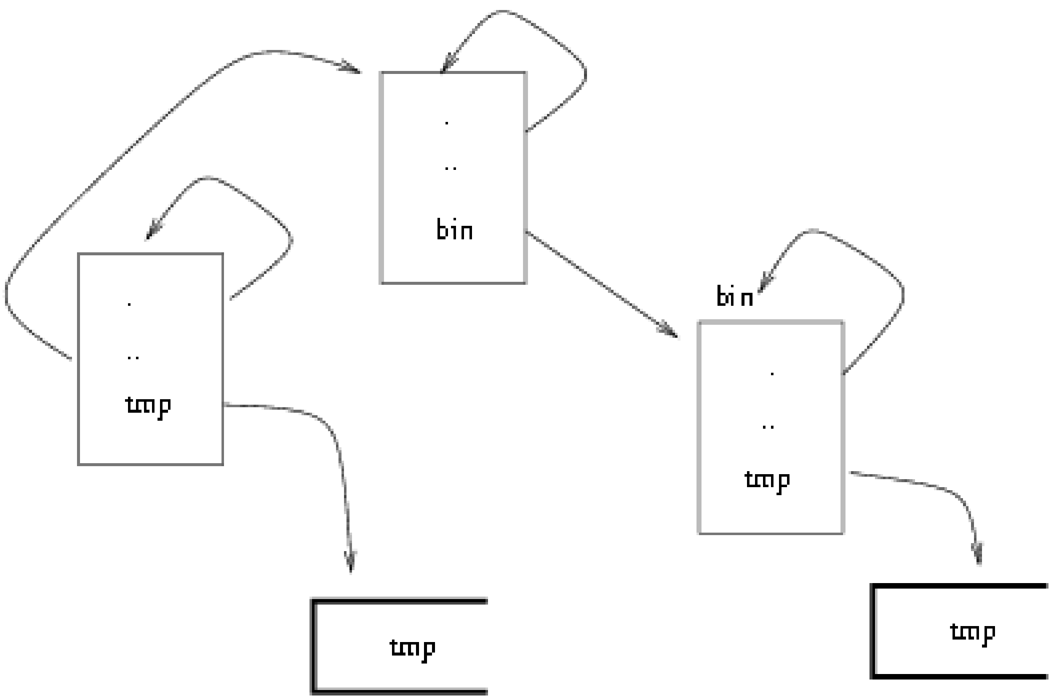
```
PROG=gros_calcul
```

```
{ echo debut de $PROG           # On opère ici
  date                          # par détournement
  $PROG                 >/dev/tty
  echo fin de $PROG             # du flux de la séquence
  date ;                } >log
```

Le programme **cp** (pour CoPy) permet de dupliquer un fichier dans un autre et de créer cet autre si besoin est. Dans tous les cas, après copie, les contenus des deux fichiers source et destination sont identiques. Attention la recopie d'un fichier dans lui-même est un moyen sûr de le remettre à vide (du fait de l'appel à **creat** sur la destination) !



La situation initiale : avant



Copie de fichiers

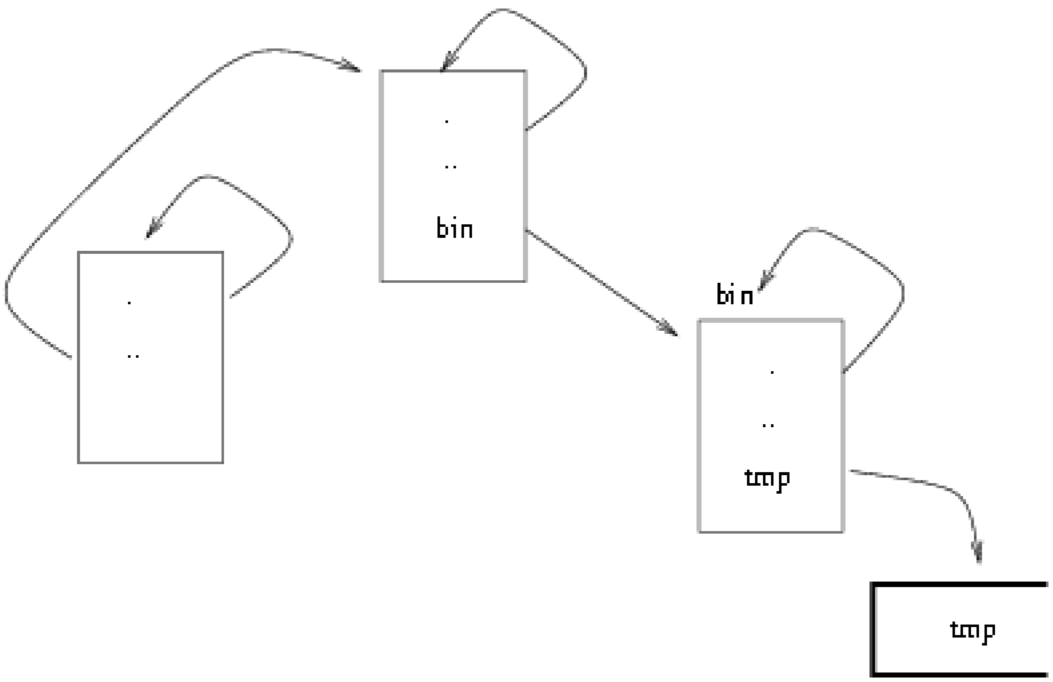
le programme **mv** (pour MoVe) permet de déplacer un fichier ou un répertoire dans l'arborescence. Pour un fichier,

`mv` x y

est grosso-modo comparable à

`cp` x y; `rm` x

mv est bien plus rapide car il ne fait que changer les liens d'attachement arboricole du descripteur de fichier. Par exemple



Déplacement de fichiers

Un important cas particulier existe lorsque répertoire de départ et d'arrivée sont identiques : il n'y a alors que changement de nom.

Le programme **ln** (pour LiNk) permet de créer des synonymes. Un fichier peut ainsi être connu sous plusieurs noms. Par exemple

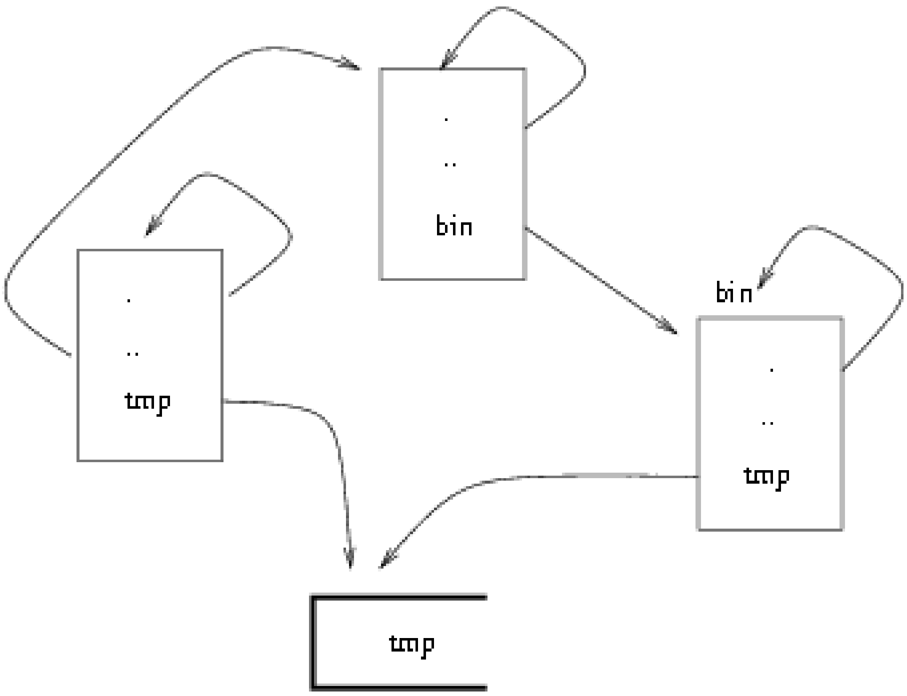
`ln sos help`

rend indiscernables **sos** et **help**. Attention toute modification de **help** ou de **sos** sera instantanée perçue de l'autre puisque ce n'est qu'un unique et même fichier. En revanche la suppression de **sos** laisse **help** inaltéré. L'analogie précédente serait mieux représentée par **ln**.

`mv` *x y*

est plus conforme à

`ln` *x y*`; rm` *x*



3.8 Les droits

Chaque fichier, répertoire ou fichier spécial est représenté par un descripteur. Entre autres choses, on y trouve

un propriétaire, une date de dernière modification, une liste de droits d'accès.

Le propriétaire d'un fichier est un utilisateur. L'ensemble des utilisateurs est scindé en groupes : ensemble d'utilisateurs travaillant, par exemple, sur un même projet. Un descripteur répartit cette masse de personnes en trois classes :

le propriétaire (unique), le groupe (non moins unique) du propriétaire, tous les autres utilisateurs.

Lecture et écriture sont les opérations fondamentales que l'on peut appliquer à un fichier. Créer ou supprimer un fichier est assimilé à une écriture dans son répertoire père.

sh ne peut exécuter un fichier que si celui-ci possède le droit d'être exécuté, c'est-à-dire si c'est un programme issu d'une édition de liens sans faute ou encore si c'est un fichier contenant un texte compréhensible par **sh**.

Pour un répertoire le sens de ces droits est un peu plus complexe. Le droit de lecture permet de consulter ce que contient un répertoire (par **ls**, par exemple). Le droit d'écriture permet de créer ou supprimer des entrées du répertoire ; enfin le droit d'exécution permet d'accéder aux entrées dont on connait les noms (surtout utile quand on ne dispose pas des deux autres droits).

Par défaut, les fichiers ou répertoires que créent un utilisateur sont dans les modes **-rw-rw-r–** et **drwxr-xr-x**. Les membres de son propre groupe ont les mêmes droits que le créateur mais les étrangers ne peuvent qu'au plus lire et fouiller mais au grand jamais détruire.

Tous les attributs d'un descripteur sont modifiables.

chown (pour CHange OWNer) change le propriétaire.

```
chown nemo *.f
```

donne à **nemo** tous les fichiers FORTRAN du répertoire courant.

chgrp (pour CHange GRouP) change le groupe.

```
chgrp daemon *.c
```

donne au groupe démoniaque les fichiers en C. Ce groupe est un groupe système comme **sys** et **bin**. **daemon** est le groupe du super-utilisateur **root**. Les utilisateurs sont souvent dans le groupe **users**.

chmod (pour CHange MODes) change les droits d'accès.

```
chmod u+xa-w * */* */*/*
```

(+**x**) rend exécutable pour le propriétaire (**u**ser) et interdit (**-**) pour tout le monde (**a**ll) le doit de modification (**w**).

On peut consulter les droits grâce à l'option **-l** de **ls**. **ls -l** liste le type, les droits, le nombre de liens, le propriétaire, la taille, la date de dernière modification de toutes les entrées du répertoire courant. Par exemple

```
$ ls -l
-rw-r--r--      1 queinnec     2205 Feb 10 20:52        u00
-rw-r--r--      1 queinnec    25932 Feb 11 22:04        u2
-rw-r--r--      1 queinnec    12699 Feb 19 21:21        u3
-rw-------      1 queinnec    12651 Feb 19 21:20        u3.CKP
drw-r--r--      1 queinnec       64 Feb  7 21:36        ufinal
-rw-r--r--      1 queinnec     2125 Feb  7 21:36        umc
-rw-r--r--      1 queinnec      988 Jan 31 21:11        unp
-rw-r--r--      1 queinnec     2398 Jan 29 22:05        unx1
```

Le type est **-** pour un fichier normal et **d** pour un répertoire. Les droits sont représentés par les neuf caractères qui suivent :

r (pour Read) est la lecture, w (pour Write) est l'écriture, x (pour eXecute) est l'« exécutabilité ».

Un tiret signale un droit non octroyé.

3.9 Les fichiers spéciaux

Les fichiers spéciaux sont des programmes particuliers que l'on peut mettre en œuvre comme s'ils étaient des fichiers ! On peut en effet les lire et les écrire par les mêmes mécanismes. C'est une des grandes forces d'UNIX que d'avoir unifié les accès aux fichiers et aux périphériques. Ecrire sur une bande magnétique ou un terminal s'énonce d'une même manière tandis qu'à l'insu de l'utilisateur est mis en œuvre, bien sûr, le programme interne adapté. Quelques fichiers spéciaux (usuellement regroupés sous /**dev**) sont **null**, **tty**, **tty***, **lp**, **hd***, **fd*** ...

3.9.1 /dev/null

/**dev**/**null** est, en écriture, un puits sans fond où s'engouffrent tous les caractères qu'on y peut écrire. En lecture c'est le flux nul ! /**dev**/**null** est pratique pour se débarasser des sorties encombrantes (messages d'erreur)

```
rm *.f 2>/dev/null
```

ne récriminera jamais.

3.9.2 /dev/tty

/dev/tty est, pour chacun, son couple clavier-écran. Le programme **/usr/bin/tty** délivre le nom physique de ce couple.

```
$ tty
/dev/tty02
```

/dev/tty ne fait, pour cet utilisateur, que rediriger ses entrées et sorties vers **/dev/tty02**. Ce dernier est l'interface entre le terminal branché sur l'ordinateur en position 2 et UNIX. Cette interface est largement détaillée plus loin.

3.9.3 /dev/lp

Parmi tous les dispositifs communs se trouve **/dev/lp** (pour Line Printer) où est branchée l'imprimante principale du calculateur. On peut imprimer un fichier en le détournant sur elle comme dans :

```
cat fichier >/dev/lp
```

Mais afin de pouvoir la mettre en commun il est raisonnable d'utiliser **lpr** (pour Line PRinter). Ce dernier sérialise les impressions et est capable de les reprendre en cas d'indisponibilité de l'imprimante (en jargon *spooler*). On écrira alors

```
lpr fichier
```

On utilisera par ailleurs le filtre **pr** (pour imPRimer) qui prépare les sorties en découpant le listage en pages, en insérant en tête de chaque page le nom du fichier , le numéro de page et la date. L'écriture standard est alors

```
print fichier
```

où **print** est le script

```
pr $* | lpr
```

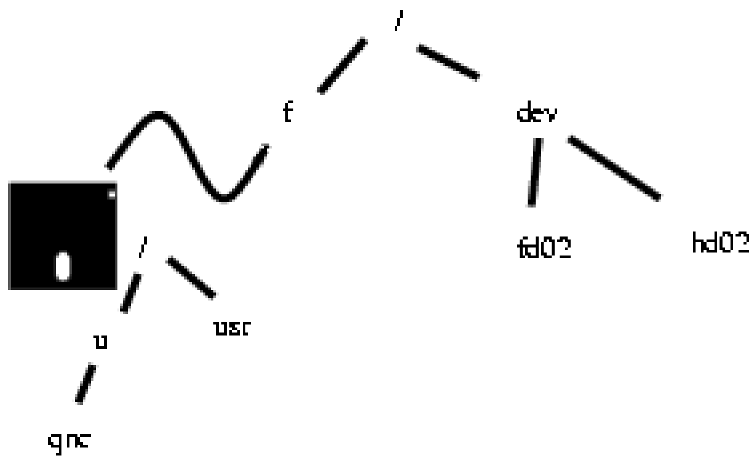
3.9.4 Les supports magnétiques

Les fichiers sont stockés sur disques durs **hd ? ?** (pour Hard Disk numéro quelquechose) ou sur disquettes **fd ? ?**(pour Floppy Disk numéro quelquechose). Les **/dev/ ?d ? ?** sont les interfaces vers ces périphériques de mémorisation.

Ces périphériques peuvent être montés ou démontés. Un periphérique est monté lorsqu'il est physiquement présent et qu'UNIX le sait. Par exemple, insérer une disquette dans le lecteur la rend physiquement présente. Ordonner

```
mount /dev/fd02 /f
```

indique à UNIX, que l'arborescence présente sur **fd02** est à greffer sur l'arborescence principale sous le répertoire **/f**.



On peut bien sûr choisir librement n'importe quel autre répertoire (même /) pour y greffer /**dev**/**fd02**. On peut démonter un périphérique grâce à

```
umount /dev/fd02
```

puis en l'ôtant physiquement.

3.10 Repères

La tradition a fixé l'usage d'un grand nombre de fichiers et de répertoires. Cette arborescence est décrite en **HIER(7)** dans le manuel de Référence UNIX. Voici cependant quelques points de repères.

/**bin** les binaires des programmes natifs d'UNIX.

/**dev** les interfaces vers les périphériques.

/**tmp** contient des fichiers temporaires de travail. Ce répertoire est régulièrement nettoyé de tout ce qui s'y trouve. Si vous avez besoin instantanément d'espace, placez vous sous /**tmp**.

/**usr**/**bin** D'autres binaires bien utiles.

/**usr**/**ucb** encore d'autres utilitaires en provenance de l'Université de Californie (Berkeley) qui est un grand centre de développement UNIX.

/**usr**/**lib** des bibliothèques (*LIBrary*) pour les utilitaires comme **nroff** .../**usr**/**local** des utilitaires propres au site.

/**etc** tout ce qui est crucial pour le système : tables des utilisateurs (**passwd**), des groupes (**group**), des périphériques montés (**mtab**) ...

Chapitre 4

Éditeur de textes

4.1 Introduction

Un éditeur de textes (piètre traduction de *text editor*) est un programme permettant la manipulation de textes, c'est-à-dire l'ajout, la modification, la suppression, le déplacement de chaines de caractères ...Ce manuel a, par exemple, entièrement été saisi au moyen d'un éditeur de textes. C'est un très important programme puisqu'un programmeur passe le plus clair de son temps devant (plus de 75

Les éditeurs se partagent en trois classes suivant la vision qu'ils offrent du texte édité

ligne	(unidimensionnel)
écran ou pleine page	(bidimensionnel)
multifenêtre	(tridimensionnel)

Les plus anciens (par exemple **ed**) sont du premier type. Les interactions se font principalement au sein d'une ligne sur laquelle on est positionné. On n'a jamais qu'une vision toujours partielle du texte édité et le plus souvent restreinte à une seule ligne.

Les nouveaux (**vi**, **screen**, **emacs**...) sont dits « pleine page ». Ils obéissent au grand principe suivant : On ne voit sur l'écran que ce qui se trouve dans le texte édité, toute modification apportée au texte est donc immédiatement visualisée sur l'écran. On développe par là même une édition de textes par essais/erreurs et donc un apprentissage aisé puisque l'on perçoit toujours ce que l'on altère.

vi est l'éditeur classique d'UNIX, **emin** est une version d'**emacs** réalisée en France par G.Berry (Ecole des Mines, Sophia-Antipolis). **emacs** est un merveilleux éditeur distribué sur de nombreuses machines et sous de nombreux systèmes (DEC, HIS ...)

La troisième classe procure la possibilité de scinder l'écran physique en de multiples fenêtres (**emacs**, **winnie** (**winnie** est dû à P.Amar (Non Standard Logic).) ...) permettant ainsi de nombreuses vues simultanées sur plusieurs textes. Il est alors courant d'avoir le texte à mettre au point dans une fenêtre, sa compilation dans une seconde, et sa mise au point dans une troisième, enfin, de virevolter de l'une à l'autre suivant les problèmes.

4.2 Caractéristiques

Les éditeurs ont de nombreux points communs.

Un éditeur manipule un ou plusieurs tampons. Les tampons portent un nom et servent de copie de travail pour les fichiers édités. Lorsque le tampon est correct, on écrase, avec son contenu, le fichier initial. Une fenêtre est associée à un unique tampon et se définit par sa taille (longueur et largeur) et son emplacement sur l'écran. A tout moment une seule fenêtre est active : c'est la fenêtre courante et toutes les commandes d'édition (sollicitées dans une fenêtre nommée *minibuffer* et réduite à une seule ligne) sont relatives au tampon qu'elle visualise.

4.3 Les commandes d'édition

Elles se séparent en plusieurs classes.

Les commandes de déplacement par caractère, mot, ligne, phrase, paragraphe, fenêtre ...ainsi que les recherches sur simples chaines de caractères ou expressions régulières (c'est-à-dire des expressions usant de méta-caractères de façon similaire à l'expansion). La syntaxe des expressions régulières de **ed**, quoique lourde mais complète, est commune à **sed** et **grep**.

Les commandes de modification : effacement de caractères, mots, lignes, phrases, paragraphes ...les insertions de caractères ou de fichiers, les substitutions multiples de chaines de caractères entre elles.

Les commandes relatives aux tampons : création, rangement de tampon, écriture de tampon dans un fichier, destruction d'un tampon.

Enfin des commandes d'environnement comme : appeler UNIX depuis l'éditeur, lire son courrier sous l'éditeur, invoquer **make** etc.

On trouvera en Annexe 3 les principales commandes communes à **emin**, **winnie** et **emacs**.

4.4 Les variantes

Les éditeurs lignes usent de deux modes : le mode d'édition où l'on soumet des commandes d'édition et le mode d'insertion où tout ce qui est frappé est inséré dans le texte (ce qui nécessite qu'une séquence particulière de caractères soit considérée comme terminant l'insertion). Les éditeurs plein écran soit respectent cette philosophie (**screen**, **vi** ...) soit sont systématiquement en mode d'insertion (**emin**, **emacs** ...). Toute séquence non reconnue comme étant celle d'une commande à exécuter est insérée sous le curseur.

Les éditeurs peuvent être extensibles (**emacs**, **winnie**). On peut programmer de nouvelles fonctions que l'on peut associer à certaines séquences de caractères. Le langage dans lequel sont écrites ces extensions est souvent LISP.

Enfin les éditeurs peuvent être structurels (ou syntaxiques) comme Mentor. Ils ont une connaissance parfaite du langage de programmation dans lequel est rédigé le texte à éditer. Il est alors impossible de créer un texte syntaxiquement incorrect !

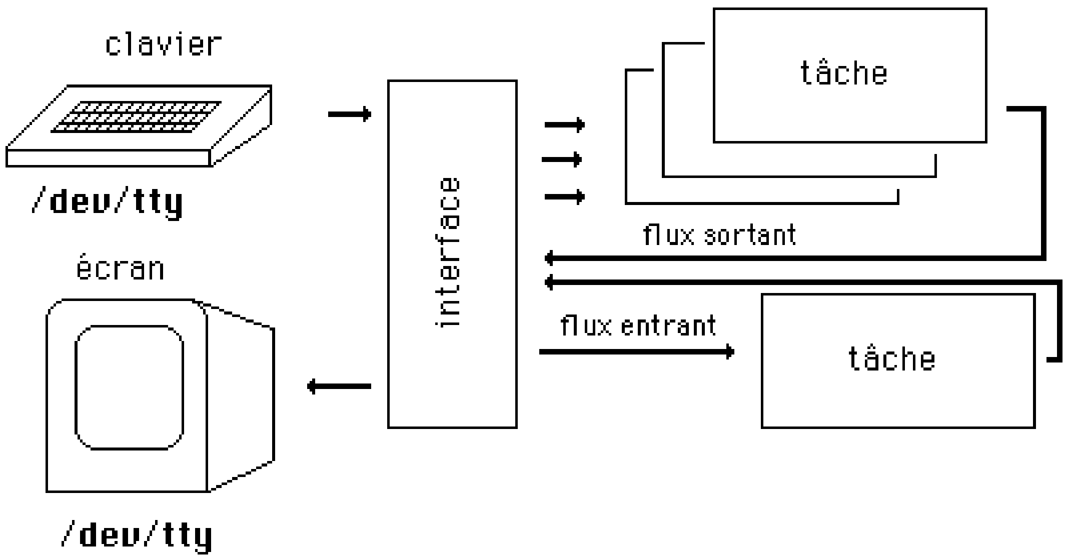
Chapitre 5

Interface

5.1 Introduction

L'interface entre machine et utilisatrice/teur est supposé constituée d'un clavier et d'un écran (c'est maintenant la combinaison la plus courante). Le clavier sert à émettre des caractères vers la machine, l'écran visualise les réponses. Les chapitres précédents ont montré comment les tâches pouvaient requérir des flux en entrée et comment l'on pouvait détourner des flux de sortie vers l'écran. Clavier et écran sont connus comme étant le fichier spécial **/dev/tty**.

En fait, tout caractère frappé au clavier n'est pas nécessairement transmis à la tâche le requérant. S'intercale en effet, entre clavier et tâches, une interface interprétant tous les caractères entrant et sortant.



Certains caractères particuliers provoquent des traitements au niveau de l'interface tandis que les autres sont seulement transmis. Une commande (**stty** pour Set TeleTYpe) et des appels au noyau UNIX (**ioctl** pour Input/Output ConTroL) permettent de gérer cette interface.

L'interface, aussi appelée *handler*, est le programme le plus proche (au sens physique) du couple clavier-écran. Ce n'est pas une tâche et les caractères qu'elle reçoit directement du clavier ou émet non moins directement sur l'écran ne forment pas des flux. Ce n'est au sens d'UNIX, qu'un fichier spécial : **/dev/tty**. Ce fichier est « doublement » spécial puisque, bien que son nom soit unique, il désigne pour toute utilisatrice/teur son propre couple clavier-écran.

5.2 La normale

Le mode natif de l'interface est celui où l'on se trouve après connexion (**login**), il peut bien sûr être modifié au goût de chacun dans **.profile** si l'on utilise **sh** ! L'alphabet (majuscule ou minuscule), les chiffres et quelques autres caractères comme les symboles de ponctuation sont, sitôt frappés, rangés dans un tampon (la ligne courante) et réémis, à titre d'écho, sur l'écran.

Les commandes

```
stty -echo # supprime l'écho
```

et

```
stty echo  # le rétablit
```

permettent de contrôler cet écho. Ainsi l'entrée du mot de passe en début de session s'effectue-t'elle sans écho pour des raisons évidentes de confidentialité.

5.3 De l'intelligence des terminaux

L'alphabet ASCII (pour American Standard Code for Information Interchange) se subdivise en deux grands sous-ensembles :

- Les caractères visualisables
- les caractères de contrôle

Les caractères de contrôle occupent les positions 0 à 31 et 127 (cf. Annexe 1). Les terminaux visualisent ou impriment les caractères normaux (à quelques traductions près sur certaines imprimantes françaises) à l'endroit précis où se trouve le curseur. Celui-ci se déplace alors d'un caractère vers la droite. Le curseur désigne toujours l'espace où insérer le prochain caractère sur l'écran.

Les fonctions propres que possèdent les terminaux intelligents sont programmées grâce aux caractères de contrôle. Par exemple, la plupart des écrans lorsqu'ils recoivent le caractère ↑ **L** positionnent le curseur en haut et à gauche (en jargon et sur certains claviers *home*). D'autres effacent l'écran en même temps. Les terminaux respectant la norme ANSI (avec ANSI, American National Standards Institute) (issue de la famille VT100 (VT100 est une marque déposée par DEC) de DEC (Digital Equipment Corporation) requièrent pour cette même fonction la séquence ESC (c'est-à-dire ↑ **[**) suivi de [, 2 et J.

Les fonctions les plus banales sont

- ↑ **H**, Backspace ou Espacement Arrière Déplace le curseur d'un caractère vers la gauche. L'effet est variable si ordonné en début de ligne !
- ↑ **M**, Carriage Return ou Retour Chariot Déplace le curseur en début de ligne.

Ces deux fonctions n'altèrent que la position du curseur à l'exception des caractères visualisés.

- ↑ **J**, Line Feed Déplace, sans changer de colonne, le curseur à la ligne suivante. Cette fonction, appelée lorsque le curseur se trouve sur la ligne du bas, provoque la disparition de la première ligne de l'écran et corrélativement l'apparition en bas d'une nouvelle ligne blanche sur laquelle se positionne le curseur.

Nous ne supposerons, pour lors, que ces trois seules fonctions.

5.4 La ligne courante

La notion de ligne courante n'existe que parce que l'on commet des erreurs de frappe. Ces erreurs sont correctibles tant que l'on demeure au sein d'une même ligne. Deux caractères spéciaux permettent d'éditer (c'est-à-dire de corriger ou plus généralement de modifier) la ligne courante.

5.4.1 Effacement de caractère

Ce caractère (*erase character* en terminologie UNIX.)

annule le dernier caractère présent dans la ligne courante. Visuellement, sur les lignes de transmission rapide (où quelques caractères de plus ne se font aucunement cruellement sentir), l'écho est ↑ **H**, SP (SP est le code ASCII numéro 32, ou blanc. Les caractères seront nommés indifféremment par leurs divers noms. SP est plus commode que « » !), ↑ **H** ce qui permet de revenir en arrière d'un caractère, de l'effacer en écrivant par dessus un blanc, puis de se repositionner sur ce même caractère effacé. Le prochain caractère frappé au clavier remplacera donc celui-là même qui vient d'être effacé.

Schématiquement (le symbole _ représente, dans la ligne courante, l'endroit où insérer le prochain caractère et sur l'écran, la position du curseur.)

caractères émis par le clavier	état du tampon	caractère émis vers l'écran	état de l'écran
initialement	_		_
w	w_	w	w_
h	wh_	h	wh_
↑ H	w_	**↑ H**	wh
		SP	w _
		↑ H	w_

Les choses ne sont pas si simples sur une imprimante où l'on ne peut effacer ce qui a été imprimé. On peut alors préférer, comme dans les vieux UNIX effacer avec # plutôt que **↑ H**. L'écho est alors limité au seul caractère # ce qui permet par simple inspection visuelle de déduire ce qui reste de la ligne courante.

```
wha#oiz##is quie#i##einnec
```

est alors la représentation, non équivoque quoique illisible, de

```
whois queinnec
```

Ce processus est répétitif, plusieurs frappes consécutives de **↑ H** effacent successivement les derniers caractères entrés. On ne peut bien sûr effacer plus de caractères qu'il n'y en a dans le tampon. Schématiquement

caractères émis par le clavier	état du tampon	caractère émis vers l'écran	état de l'écran
initialement	w_		w_
↑ H	_	**↑ H**	w
		SP	_
		↑ H	_
↑ H	_	rien	_

Le caractère d'effacement peut être programmé par

```
stty erase 'c'
```

où *c* peut être un caractère quelconque même visualisable comme dièse.

```
stty erase '#'
```

ou encore un caractère de contrôle exprimé comme une séquence de deux caractères dont le premier est l'accent circonflexe, et le second un caractère visualisable (cf. Annexe 1). Par convention, dans tout ce texte, toute séquence ***↑ x*** désignera un caractère de contrôle tandis que $\hat{}\,x$ ne sera que la suite des caractères accent circonflexe et lettre *x*. Cette dernière séquence permet de représenter pour certains programmes comme **stty** un caractère de contrôle de façon lisible. On écrira :

```
stty erase '^H'
```

ou même

```
stty erase '^h'
```

Une troisième solution consiste à faire précéder BS d'un caractère d'échappement approprié. par exemple, pour **sh** (cf. Désinformation locale)

`stty erase '`***↑ V↑ H***`'`

Enfin si l'on veut supprimer cette fonction d'édition, il suffira pour cela d'ordonner

```
stty erase undef
```

ou encore

```
stty erase u
```

où **undef** et **u** signifient *undefined*. Dans ce dernier cas ↑ **H** est considéré comme un caractère normal et donc transmis et réémis en écho.

caractères émis par le clavier	état du tampon	caractère émis vers l'écran	état de l'écran
initialement	wh_		wh_
↑ **H**	wh↑ **H**_	↑ **H**	w$\underline{\text{h}}$

La ligne qui sera fournie à la tâche qui a demandé cette lecture sur le clavier comportera cet espacement arrière en troisième caractère.

5.4.2 Annulation d'une ligne entière

Effacer complètement la ligne courante parce que l'on a commis une faute en début de ligne ou bien que l'on a changé d'idée, est possible grâce à ↑ **X** (ou *kill character* en terminologie UNIX).

En écho est émis ce qu'il faut comme séquence ↑ **H**, SP, ↑ **H** pour effacer toute la ligne présente sur l'écran. On peut bien sûr programmer ce caractère grâce à

```
stty kill '^X'
```

ou comme dans les vieux UNIX

```
stty kill '@'
```

On peut bien sûr annuler cette fonction par

```
stty kill undef
```

5.5 Désinformation locale

Si l'on peut maintenant commodément éditer la ligne courante, il n'est malheureusement plus possible d'émettre ↑ **H** ou ↑ **X** vers une tâche. Un nouveau mécanisme est nécessaire : l'échappement ! Il permet de considérer le caractère qui suit, quelque soit celui-ci, comme un caractère normal. L'échappement est un mécanisme universel que l'on retrouve dans de nombreux programmes interactifs

- \ pour **sh**, **ed**, **nroff** ...
- ↑ **Q** pour les éditeurs pleine page (**emacs**, **emin** ...),
- ↑ **V** pour l'interface.

Ainsi pour transmettre un espacement arrière sans que celui-ci soit interprété comme l'effacement du dernier caractère de la ligne courante, le fera-t'on précéder du caractère d'échappement. Par exemple

caractères émis par le clavier	état du tampon	caractère émis vers l'écran	état de l'écran
initialement	who_		who_
↑ **V**	who_	rien	who_
↑ **H**	who↑ **H**_	↑ **H**	wh$\underline{\text{o}}$

mais si l'on continue

caractères émis par le clavier	état du tampon	caractère émis vers l'écran	état de l'écran
↑ **H**	who_	↑ **H**	who
		SP	w o
		↑ **H**	w_o

Le premier ↑ **H** précédé du caractère d'échappement a été pris tel quel tandis que le second a été interprété ce qui ramène la ligne courante à sa valeur initiale. Notez cependant l'altération de la visualisation de la ligne courante.

On peut programmer le caractère d'échappement par

```
stty lnext '^V'
```

où **lnext** signifie *literal-next* : le caractère suivant est à prendre comme une lettre (un caractère normal).

5.6 Voir l'invisible

Les caractères de contrôle ne sont pas directement visualisables. Ils peuvent l'être par leurs effets sur le curseur ou l'écran (et encore seulement à basse vitesse). On peut aussi les associer à des séquences de caractères normaux. Par exemple Retour Chariot peut être associé à

- CR, dans ce présent ouvrage et selon la terminologie ASCII (cf. Annexe 1)
- ↑ **M**, UNIX qualifie cette notation de *à la DEC*
- `\015`, c'est la vieille notation octale propre à UNIX et C, aberrante sur des machines à 8/16/32 bits mais maintenue par une tradition décennaire.

L'interface peut être programmée pour visualiser les caractères de contrôle émis en écho

```
stty ctlecho # le mode natif
```

ou pour supprimer cet écho particulier

```
stty -ctlecho
```

Dans ce mode, l'exemple précédent conduira à :

caractères émis par le clavier	état du tampon	caractère émis vers l'écran	état de l'écran
initialement	who_		who_
↑ **V**	who_	rien	who_
↑ **H**	who↑ **H**_	ˆ	whoˆ_
		H	whoˆH_
↑ **H**	who_	↑ **H**	whoˆH
		SP	whoˆ _
		↑ **H**	whoˆ_
		↑ **H**	whoˆ
		SP	who _
		↑ **H**	who_

L'interface use toujours de caractères de contrôle mais visualise différemment la ligne courante. Quoique représenté graphiquement par deux caractères, l'espacement arrière est traité comme il doit l'être : comme un unique caractère.

Il n'est plus possible de se rendre compte du contenu exact de la ligne courante, sauf à très basse vitesse où l'on peut suivre les mouvements du curseur. **ctlecho** (pour ConTroL character ECHO) ne s'intéresse qu'au seul écho des caractères entrants. Les caractères sortant ne sont nullement affectés et pour visualiser ceux-ci, utilisez, par exemple, **cat -v -t -e**.

Parmi les caractères de contrôle, la tabulation (↑ **I** ou TAB) joue un rôle particulier. Tout comme les machines à écrire, et dans un même but de composition de tableaux à colonnes soigneusement alignées, la tabulation lorsque frappée au clavier enjoint au curseur de se placer sur la prochaine tabulation. Sur les écrans celles-ci sont généralement placées toutes les huit positions à partir de la première colonne. Sur les imprimantes, elles sont souvent réglables à des valeurs quelconques. Pour les dispositifs ignorant la tabulation l'interface peut simuler cette fonction. TAB, en écho, est convertie en le nombre de SP nécessaire pour que le curseur regagne la prochaine tabulation. Ainsi donc

caractères émis par le clavier	état du tampon	caractère émis vers l'écran	état de l'écran
initialement	who_		who_
↑ **I**	who↑ **I**	SP	who _
		SP	who _
		SP	who _
		SP	who _

la tabulation a ici engendrée quatre blancs pour amener le curseur en neuvième colonne. Un seul Retour Arrière annule ces quatre blancs.

caractères émis par le clavier	état du tampon	caractère émis vers l'écran	état de l'écran
initialement	who↑ **I**_		who _
↑ **H**	who_	↑ **H**	who _
		↑ **H**	who _
		↑ **H**	who _
		↑ **H**	who_

5.7 Émissions diverses

Tout ce que l'on a vu précédemment permet de s'assurer de la correction de la ligne courante. On peut forcer l'envoi de cette ligne à la tâche requérante grâce au caractère ↑ **J** ou LF ou Line Feed. le tampon est alors effacé et le curseur saute à la ligne suivante au moyen de la séquence LF, CR. Le caractère LF est lui-même transmis comme dernier caractère de la ligne courante. Schématiquement

caractères émis par le clavier	état du tampon	caractère émis vers l'écran	état de l'écran
initialement	date_		date_
↑ **J**	_	↑ **M**	date
	et	↑ **J**	
	envoi de		date
	date↑ **J**		_

Sur la plupart des claviers la grande touche à droite marquée RETURN, RETOUR ou encore ENVOI est associée au code CR c'est-à-dire ↑ **M** plutôt qu'à LF. On peut alors programmer l'interface pour que soit converti, en entrée, CR en LF !

```
stty -nl
```

Symétriquement en sortie, LF est converti en CR, LF. Ainsi donc, dans ce mode qui est natif

caractères émis par le clavier	état du tampon	caractère émis vers l'écran	état de l'écran
initialement	date_		date_
↑ **M**	_	↑ M	<u>d</u>ate
	et	↑ **J**	
	envoi de		date
	date↑ **J**		_

Inversement

```
stty nl
```

où ↑ **M** est un caractère normal et où seul ↑ **J** force l'envoi d'une ligne, conduirait au dialogue

```
$ who↑ J
      queinnec
              $ who↑ M↑ J
: program not found
```

Les deux dernières lignes demandent explication. CR étant un caractère normal et **who↑ M** n'étant le nom d'aucun programme, le message d'erreur est donc

```
who↑ M : program not found
```

ce qui explique la visualisation donnée plus haut, puisqu'en sortie CR n'est pas converti en *^M* mais est interprété par l'écran comme une commande de positionnement de curseur. UNIX nomme *newline* (ou *NL*)le caractère LF : c'est le délimiteur de ligne dans les fichiers UNIX, c'est aussi un séparateur syntaxique pour **sh**.

5.8 Au coup par coup

Un éditeur pleine page, un programme comme **more** sollicitent puis utilisent un à un les caractères frappés. Il n'y a plus de notion de ligne courante éditable, il n'y a plus qu'une lecture directe caractère par caractère. Ce mode est possible grâce à

```
stty cbreak
```

On revient au mode natif par

```
stty -cbreak
```

On peut, en C ou en d'autres langages, utiliser la fonction **getchar** qui retourne le caractère suivant du flux. l'usage de cette fonction est indépendant de la programmation de l'interface et si cette dernière est en mode **-cbreak** c'est-à-dire en mode ligne, le premier appel à **getchar** bloque le programme jusqu'à ce que l'utilisatrice/teur frappe, entre autres caractères un NL. Le programme sera alors redémarré et consommera tous les caractères frappés d'avance. Par exemple,

```
#include <stdio.h>
main () { char c; /* Le caractère courant */
while ( (c=getchar()) != EOF ) putchar(c);
exit(0); }
```

le programme-miroir précédent (en C) fait écho de chaque caractère qu'il lit. Ainsi obtiendra-t'on le comportement suivant (les sorties d'UNIX sont en gras).

```
$ stty -cbreak echo
$ miroir
On peut taper ce que l'on veut↑ J
On peut taper ce que l'on veut↑ J
mais il faut attendre la fin de ligne↑ J
mais il faut attendre la fin de ligne↑ J
```

```
↑ D
$ stty cbreak echo ; miroir ; stty -cbreak
eecchhoo !!
↑ \
Interrupt ...
$
```

Il n'y a qu'une seule interface pour votre couple écran-clavier. Toute programmation spécifique de l'interface est donc partagée par toutes les tâches qui y accèdent. La remise en état de l'interface est souvent nécessaire après meurtre impromptu.

↑ ***J***`stty nl echo -cbreak`↑ ***J***

est un bon moyen pour ce faire, mais

↑ ***J***`reset;tset`↑ ***J***

est encore meilleur. **reset** est, comme son nom l'indique, un programme qui remet une interface en mode natif ; **tset** (pour Terminal SET) réinitialise physiquement un terminal en lui émettant ce qu'il faut pour qu'il se sente en aussi bonne forme qu'après un arrêt-marche.

5.9 Extensions

De nombreuses autres possibilités existent, programmables dans l'interface.

- L'effacement du dernier mot de la ligne courante : ↑ **W**
- La visualisation de la ligne courante : ↑ **R**, surtout utile si polluée par des caractères émis par une autre tâche.
- Enfin le contrôle du flux sortant : ↑ **S** et ↑ **Q**

Lorsque deux processus informatiques sont en séquence, c'est-à-dire que le premier produit des caractères que le second consomme (par exemple, un clavier et l'interprète de commandes, ou encore une tâche et l'écran), il est souvent nécessaire de rythmer cet échange en fonction de leur vitesse propre. Il est du rôle du consommateur (ou du système s'il s'y substitue) de prévenir le producteur si ce dernier en fait trop, c'est-à-dire s'il produit plus que ne saurait en ingérer le consommateur.

Le protocole X-ON, X-OFF ou DC1, DC2 (c'est-à-dire ↑ **Q**, ↑ **S**) est une solution à ce problème. Lorsque le consommateur est en voie d'être saturé il émet ↑ **S** (S comme Stop) au producteur qui s'arrête alors de produire. Réciproquement lorsque le consommateur, précédemment saturé, est en passe de totalement rattraper son retard et risque donc d'être totalement affamé, il relance le producteur par un ↑ **Q** bien senti !

Ce même protocole peut être employé par une/ utilisatrice/teur, lorsqu'en danger de ne plus pouvoir lire ce qui s'affiche sur son écran, elle/il stoppe l'interface par ↑ **S**, lit posément ce qui est visible et relance le défilement par ↑ **Q**.

↑ **O** bloque l'écran où il en est (comme ↑ **S**) mais sans interrompre le producteur. La saturation est évitée par élimination pure et simple des caractères non affichés. ↑ **O** est un caractère agissant comme une bascule puisqu'une nouvelle frappe fait retourner dans le mode précédent. On peut ainsi échantillonner les sorties. Cependant, à haute vitesse, ces trois caractères nécessitent un doigté certain qu'un programme comme **more** permet de mettre en œuvre bien plus commodément !

5.10 Mort d'un flux

Les filtres, c'est-à-dire les tâches qui exploitent un flux pour en produire un autre, ne vivent que la durée du flux qui les alimente. Une fin de flux est déterminée par une réponse négative à une demande de lecture. Lorsqu'un flux est créé par une lecture dans un fichier (suite d'octets bornée), la fin de flux est facile à prévoir. Mais lorsque le flux provient du clavier ?

On peut ordonner à l'interface de provoquer une fin de flux, ainsi la tâche en lecture saura-t'elle que le flux issu du clavier est achevé. La commande

```
stty eof '^D'
```

permet de programmer le caractère de fin de flux. Celui-ci émis en début de ligne (c'est-à-dire juste après un LF) ou encore répété deux fois, provoque la terminaison du flux requérant. Le caractère $\uparrow$ **D** n'est pas transmis et n'est donc utilisé que localement dans l'interface. Par exemple, la commande suivante

```
{ tee </dev/tty
  echo FIN 1
  tee </dev/tty
  } >/tmp/test
```

et les caractères suivants frappés au clavier
abc$\uparrow$ **J**def$\uparrow$ **J**ghijkl$\uparrow$ **D**$\uparrow$ **D**mnop$\uparrow$ **J**$\uparrow$ **D**
créeront, dans le fichier : **/tmp/test**

```
abc
def
ghijklFIN 1
mnop
```

On peut, bien entendu, transmettre un $\uparrow$ **D** en le préfixant de $\uparrow$ **V** ! De nombreux programmes sont des filtres comme, par exemple, **tee**, **mail**, **sed**ou encore **sh**. On peut donc, si l'on est dans le bon mode d'interface, terminer une session en frappant $\uparrow$ **D** ou encore grâce à la commande intrinsèque **exit**.

5.11 Signalisation

Deux autres caractères utiles existent qui ont des effets dépassant de loin l'interface, ce sont les caractères d'interruption (**intr** pour INTeRupt) et d'abandon (**quit**). Ils se programment de façon classique

```
stty intr '^?'
```

```
stty quit '^\'
```

– ˆ ? correspond au caractère DEL (ASCII 127)
– ˆ est une notation (cf. Manipulation) pour le caractère $\uparrow$ \ (ASCII 28)

Les tâches peuvent communiquer entre elles par fichiers, par cascades mais aussi par signaux. Les deux premières façons permettent d'échanger des messages composés d'un nombre quelconque d'octets. Il est souvent nécessaire d'alerter une tâche sans pour cela lui fournir pléthore d'informations. De plus, deux tâches peuvent vouloir échanger des signaux sans pour cela ouvrir un fichier commun (solution lourde et coûteuse) ni mettre en œuvre une cascade ce qui ne se peut (sous UNIX V7) qu'entre tâches issues d'un même ancêtre ayant ouvert cette fameuse cascade ! La solution réside dans l'envoi de signaux entre tâches qui n'ont alors besoin de connaitre que leur numéro. Le signal peut être affublé d'une sémantique quelconque laissée à l'initiative des interlocuteurs.

Quelques dizaines de signaux standard existent. Les plus courants sont
– SIGINT le signal d'interruption (2)
– SIGQUIT le signal d'abandon (3)
– SIGKILL le signal de meurtre (9)

Un signal peut être envoyé à n'importe quelle tâche. chaque tâche est référencée par un numéro (entre 1 et par exemple, 32767). La tâche 1 est la tâche **init** lancée, comme son nom l'indique, à l'initialisation du système. A chaque création de tâche ce numéro est incrémenté. La commande **kill** permet d'envoyer un signal. Ainsi

```
kill -2 $$
```

émet le signal d'interruption à votre interprète de commandes courant (peut lui en chaud d'ailleurs car il sait les ignorer). **$$** est une variable prédéfinie dont la valeur est le numéro de la tâche qui interprète la commande.

Un signal peut aussi être envoyé à toutes les tâches, encore vivantes, issues de votre **login** initial. On utilise pour ce faire le numéro zéro. La commande

```
kill -9 0
```

tue sûrement toutes ces tâches ainsi que votre **sh** initial. **Init** voyant alors que votre terminal est ésseulé, le réinitialise en recréant un nouveau **login** qui vous permettra de continuer votre travail après avoir exorcisé vos démons antérieurs. Le signal de meurtre ne peut être ignoré et est donc toujours couronné de succès.

Le signal d'interruption tue en général la (ou les) tâche(s) qui le reçoit(vent). le signal d'abandon aussi, mais il force en outre un vidage mémoire de la tâche dans un fichier de nom **core** qui peut être inspecté par **adb**. La plupart des programmes interactifs bien au point rattrappent le signal d'interruption et se réinitialisent au niveau de leur boucle principale de commandes (par exemple, **Le_Lisp**, **adb**, **more** ...). Le signal d'abandon est en général laissé pour vraiment tuer ces tâches. Frapper DEL force l'interface à émettre

```
kill -2 0
```

tandis que ↑ \ provoque

```
kill -3 0
```

Ces actions, locales à l'interface, sont déclenchées dès que reçues et donc de façon totalement déconnectées et asynchrones des éventuelles lectures que pourraient commettre vos tâches courantes. C'est donc un moyen efficace de se signaler à l'attention d'UNIX. La touche DEL est souvent nommée sur les claviers ATTN ou ATTENTION.

5.12 Manipulations

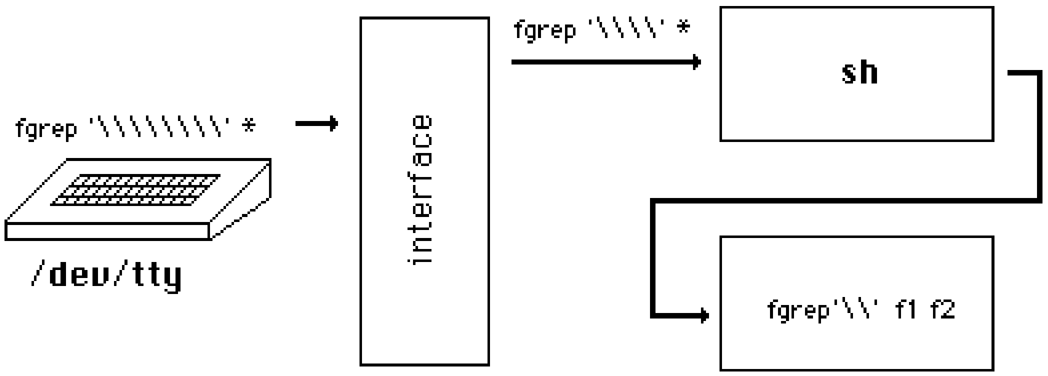
Certains programmes ont eux-mêmes une notion de ligne courante (par exemple **sh**, **Le_Lisp** et d'autres). Quoique officiant souvent en mode **cbreak** et **-echo** ils sont lancés par **sh** en mode normal. Leurs arguments d'appel passent donc à travers l'interface et **sh**. Les mécanismes peuvent donc se conjuguer. Par exemple

```
grep  '\\\\'  *
```

permet de rechercher toutes les lignes des fichiers du répertoire courant, comportant au moins une barre de fraction inverse. En effet les quatre caractères \\\\ frappés au clavier passent tels quels l'interface, sont interprétés par **sh** comme la chaine \\ (puisque \ est le caractère d'échappement de **sh**) qui est alors soumise à **grep** qui l'interprète lui-même comme l'unique caractère \, puisque \ est aussi le caractère d'échappement de **grep** ! Un bel exemple de masochisme serait alors (\ étant aussi le caractère d'échappement de **nroff** la série des huit caractères \ qui suit a été frappée sous la forme de seize \ !)

```
stty lnext '\\'
grep '\\\\\\\\\\\\\\\\' *
```

qui recherche la même chose que précédemment ! L'explication graphique est la suivante



Manipuler UNIX via TRANSPAC pose des problèmes analogues lorsque l'on désire transmettre ↑ **P**.

5.13 Lignes ?

Les fichiers UNIX sont des suites d'octets quelconques. La notion de ligne n'existe pas naturellement, ce n'est que le programme qui exploite ce fichier qui décide, s'il en a la nécessité, de son propre concept de ligne. Une convention largement suivie est qu'une ligne s'achève par un LF.

abc↑ ***V***↑ ***J***def\↑ ***J***ghi↑ ***J***

forment, au sens de l'interface, un ensemble de deux lignes

abc↑ ***V***↑ ***J***def\↑ ***J***

et

ghi↑ ***J***

Elles sont transmises à, par exemple, **sh** comme

abc↑ ***J***def\↑ ***J***
ghi↑ ***J***

sh voyant que le second LF est précédé du caractère d'échappement \ les traitera comme les deux lignes

abc↑ ***J***
def\↑ ***J***ghi↑ ***J***

ce qui aboutira à lancer le programme le programme **abc** puis le programme dont le nom est **defghi**, puisqu'une fin de ligne précédée par ẽquivaut à rien.

Dans tous les cas l'image visuelle obtenue par écho de l'interface (dans le mode **-ctlecho**) ressemblera à

```
abc
def\
ghi
```

soit trois lignes !

5.14 Exploiter des terminaux

Comme expliqué plus haut, les terminaux intelligents exploitent certains caractères de contrôle pour gérer leur écran et ainsi limiter le travail de l'unité centrale ainsi que le volume des échanges sur les lignes de communications. Les fonctions classiques sont les suivantes

- effacer tout l'écran
- effacer l'écran entre le curseur et la fin de l'écran
- effacer l'écran entre le curseur et la fin de la ligne où il se trouve
- insérer un caractère sous le curseur et déplacer d'un cran vers la droite tous les caractères suivants de la ligne
- effacer le caractère sous le curseur et ramener d'un cran vers la gauche tous les caractères suivants de la ligne
- insérer une ligne blanche au niveau du curseur et repousser ainsi toutes les lignes suivantes (en perdant celle du bas de l'écran)
- effacer la ligne sous le curseur et ajouter une ligne blanche en bas de l'écran
- positionner le curseur en un endroit quelconque de l'écran
- écrire dans un mode différent qui tranche sur le fond ambiant, par exemple clignotant, surbrillant, souligné, video inverse etc ...

De nombreuses autres fonctions existent mais seules les précédentes sont usuelles sur les terminaux actuels. En revanche leur mise en œuvre varie fortement d'un terminal à l'autre. Afin de pouvoir développer des logiciels multiterminaux, UNIX procure une bibliothèque **/usr/lib/libtermcap.a** offrant ces fonctions et dont l'interface d'appel se trouve, comme toutes les autres interfaces d'appels au système, dans le répertoire **/usr/include**, sous le nom **curses.h**.

Pour réaliser l'effet demandé ces fonctions consultent un fichier **/etc/termcap** (pour TERMinal CA-Pabilities), véritable base de données, humainement lisible, sur les terminaux actuels. Elles en déduisent les séquences de caractères appropriées au terminal visé. Le type de ce dernier est la valeur de la variable **TERM** de **sh**. Modifier gratuitement cette valeur serait funeste pour les éditeurs pleine page et autres programmes usant d'effets video.

5.15 La commande stty

La commande **stty** permet de modifier les paramètres de l'interface. Comme beaucoup d'autres commandes UNIX, employée sans arguments, elle délivre les principaux paramètres régissant l'interface.

```
stty everything
```

permet de tout savoir ! Par exemple

```
$ stty everything
new tty, speed 38400 baud
-raw nl -echo -lcase -tandem tabs -cbreak
-crtbs -crterase -crtkill -ctlecho -prterase -tostop
-tilde -flusho -mdmbuf -litout -nohang
-pendin -decctlq -noflsh
erase  kill   werase rprnt  flush  lnext  susp   intr   quit   stop   eof
^H     ^X     ^W     ^R     ^O     ^V     ^Z/^Y  ^C     ^
~    ^S/^Q  ^D
```

La commande

```
stty
```

ne renseigne que superficiellement. Par exemple

```
$ stty
new tty, speed 38400 baud; nl -echo tabs
```

Chapitre 6

Presque le fin mot !

6.1 Introduction

Le but de ce chapitre est de vous permettre d'appréhender l'essence des tâches. Il ne constitue ni une introduction à C (cf. bibliographie), ni un exposé des parties 2 et 3 du Manuel UNIX (appels systèmes et librairies d'utilitaires). Les appels que nous détaillerons sont **fork**, **exec** et **pipe**. Ces trois appels sont les primitives les plus originales d'UNIX.

6.2 Les appels systèmes

Ce sont les points d'entrée au cœur d'UNIX. Pour la plupart ils peuvent être directement mis en œuvre par l'interprète de commandes (**sh**). Par exemple,

```
sleep 3
```

correspond en C à

```
sleep(3);
```

et

```
exec /bin/sh -i fichier
```

à (toujours en C)

```
execl("/bin/sh","/bin/sh","-i","fichier",0);
```

Enfin

```
>log
```

est analogue (encore en C) à

```
creat("log",0755);
```

Un programme comme **stty** permet de mettre en œuvre tous les types d'appels exprimables directement en C par **ioctl**. par exemple,

```
stty -echo
```

est analogue à

```
/* Lire les paramètres actuels  */
ioctl(0,TIOCGETP,sgttyb0);
/* Modifier le mode écho        */
sgttyb0.sg_flags &= ~ECHO;
/* Imposer à l'interface la
   nouvelle table modifiée      */
ioctl(0,TIOCSETP,sgttyb0);
```

6.3 exec

exec, lorsque appelée par une tâche, change le programme support de cette tâche. C'est une transmutation et non une métempsychose ! En effet, on utilise la vie de la tâche qui invoque **exec** pour lui faire faire autre chose sans pour cela la tuer.

Une tâche est l'association d'un programme à exécuter et d'une vie, c'est-à-dire d'un droit à disposer de temps d'unité centrale pour y être exécutée. Un appel à **exec** en quelque partie que ce soit du programme appelant réinitialise la tâche pour lui faire exécuter un nouveau programme dont le nom est donné en argument. Ce nouveau programme débute, bien sûr, par sa première instruction et bien qu'**exec** soit un sous-programme, le programme appelant sera purement et simplement oublié comme s'il n'avait jamais existé et jamais invoqué **exec**. Quand le nouveau programme sera achevé (à condition qu'il n'appelle pas lui-même **exec**) la tâche associée sera également achevée. **exec** est l'acte magique par excellence : il transmue une citrouille en un carosse ou un ogre en une souris !

Cette sémantique est très liée à la primitive suivante.

6.4 fork

fork duplique une vie. C'est un clonage ! Une tâche qui invoque **fork** crée par scissiparité une seconde tâche en tout point semblable à elle-même, à un unique détail près ! **fork** retourne à la mère le numéro de la tâche fille engendrée mais ne retourne que zéro à la fille qui ignore donc, véritable orpheline, qui est sa mère ! Les deux tâches continuent alors leur exécution en séquence après l'appel à **fork()**,et en parallèle entre elles. Pour rester dans le domaine féérique, **fork** est analogue au coup de hache que donne l'apprenti sorcier [Goethe] au balai récalcitrant. Il ne fait qu'obtenir deux balais rigoureusement identiques et programmés pour le même travail.

fork est le seul moyen de créer une tâche. L'aïeule commune à toutes est **init** créée ex nihilo au lancement du système. Pour lancer une tâche sur un programme on utilise un **fork** suivi d'un **exec**. Comme ce qui suit :

```
#define ERREUR_FORK -1
fille=fork();           /* fourcher !   */
/* Il y a maintenant deux tâches qui
   exécutent ce qui suit                */
if ( fille != ERREUR_FORK )
   /* le clonage s'est bien déroulé !   */
   if ( fille == 0 )
      /* Seule la fille exécute ce qui suit     */
      execl("programme", ... arguments ...);
      /* et elle continuera par la première
         instruction de programme           */
   /* Par contre ici ne peut passer que la tâche mère
      qui imprime le numéro de sa fille (en décimal) */
   printf(" ma fille est : %d",fille);
   ... suite ...
```

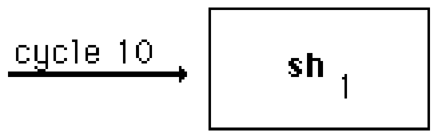
Au niveau de l'interprète de commandes, existent **&** et **exec** qui mettent en jeu les mêmes mécanismes. Donnons un exemple plus réel montrant l'énorme consommation de tâches de **sh**. Soit le fichier exécutable **cycle** contenant :

```
programme &
{ sleep $1
  kill -9 $?
  exec cycle `expr $1 + 10`
  }
```

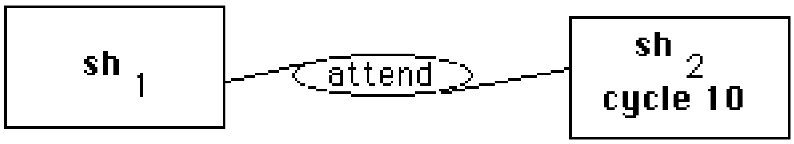
et soit **sh**⟨1⟩ (tous les noms de tâches seront indicées par le numéro de tâche associée afin que l'on puisse suivre ces mêmes tâches se transmutant et différencier celles qui ont un même programme support.)

l'interprète de commandes auquel vous soumettez

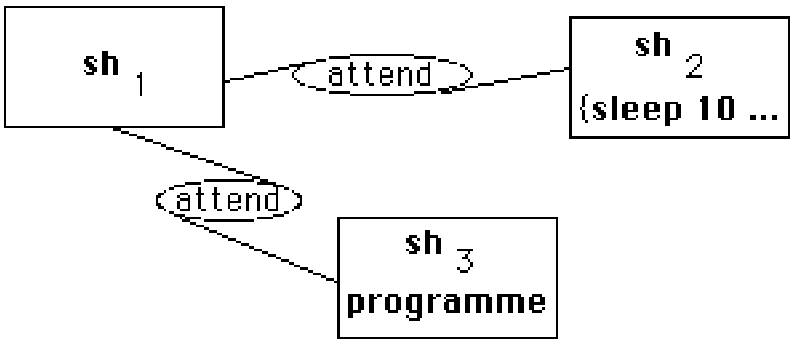
```
cycle 10
```



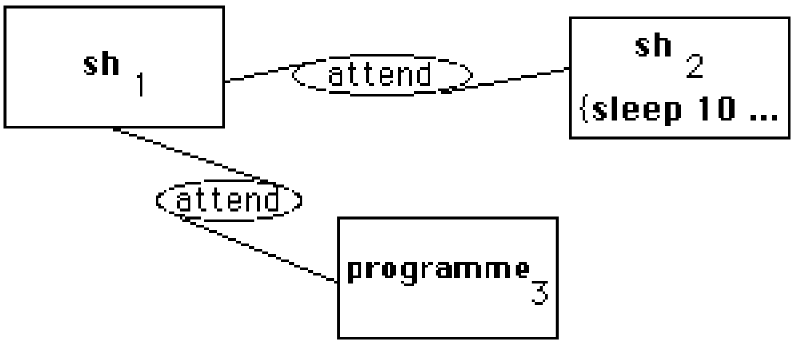
sh<1> voit que **cycle** est un programme à lancer aussi fourche-t'il (créant ainsi **sh**<2>), **sh**<2> analyse **cycle**, et **sh**<1> se met en attente de la fin de **sh**<2>.



sh<2> substitue textuellement **$1** en **10** dans tout **cycle** puis remarquant la présence d'un **&**, fourche en **sh**<3> qui analyse **programme** tandis que, sans attendre, **sh**<2> poursuit l'analyse du reste de **cycle**.



sh<3> n'étant qu'un simple programme à lancer, se transmue en **programme** grâce à **exec**.



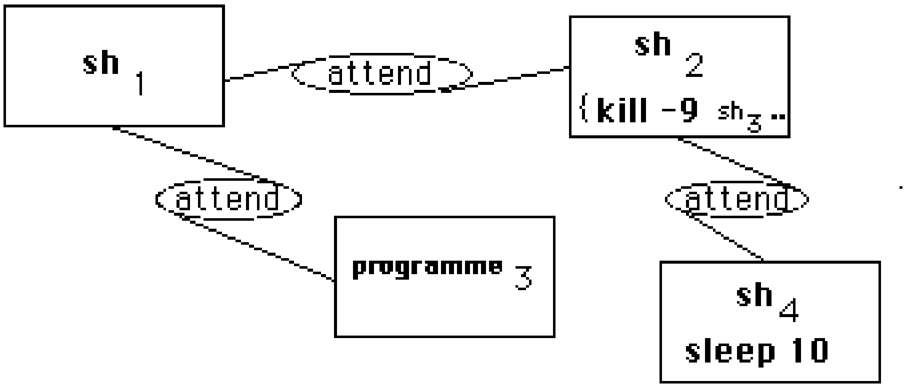
Pendant ce temps **sh**<2> substitue à **$?** le numéro de tâche de **sh**<3> (qui est aussi celui de **programme**<3> après l'**exec**), constate que c'est une séquence d'appels, fourche donc, en conséquence et en **sh**<4>, destiné à analyser

```
sleep 10
```

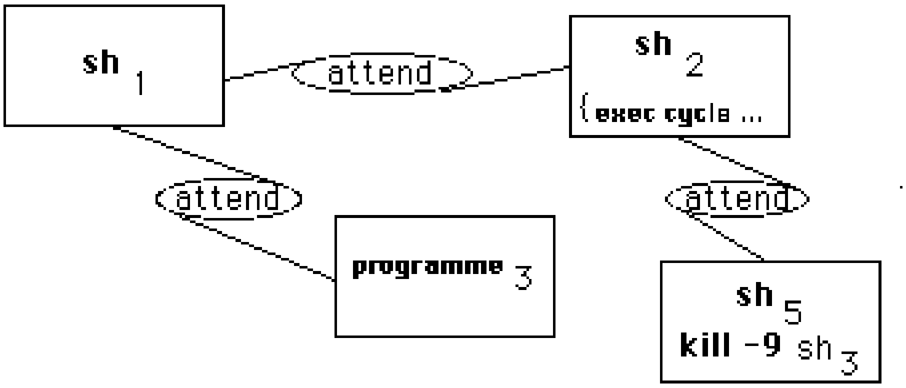
tandis que **sh**<2> diffère l'examen de

`kill -9` *sh<3>*
`exec cycle 'expr 10 + 10'`

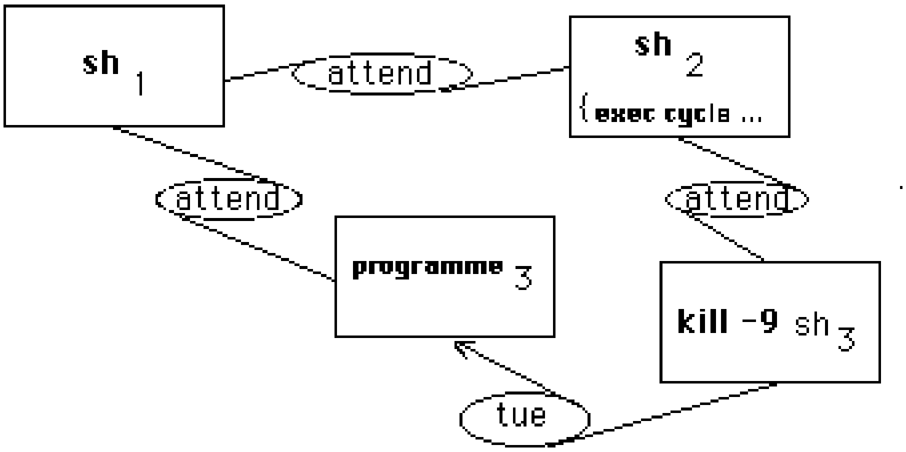
en attendant la fin de **sh**<4>.



sh<4> **exec**ute **sleep**, attend 10 secondes puis meurt, relançant ainsi **sh**<2> qui poursuit son analyse, constate que c'est une séquence et fourche donc derechef.



sh<5> examine **kill -9***sh*<3>, **exec**ute **kill**, envoie donc le signal 9 (SIGKILL) à **programme**<3> et donc le tue radicalement, sans autre forme de procès.



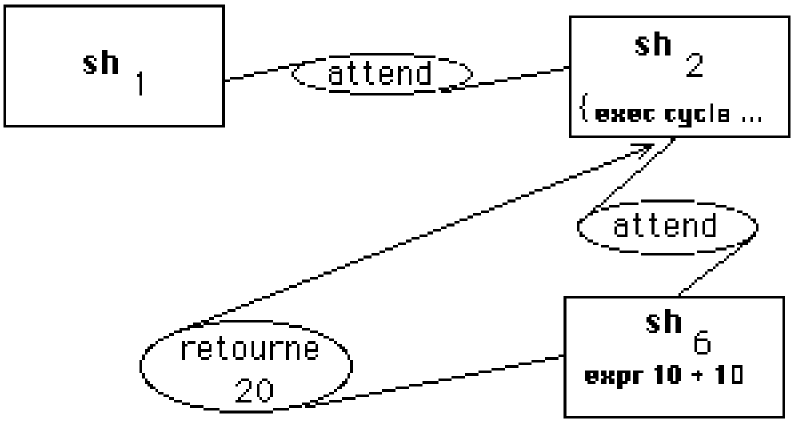
Son office achevé **sh**$<5>$ meurt. **sh**$<2>$, qui n'attendait que cela, scrute la ligne

```
exec cycle ‘expr 10 + 10‘
```

Cette commande nécessite de calculer

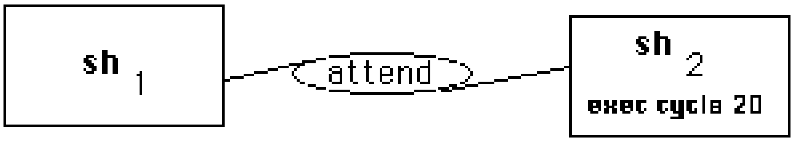
```
expr 10 + 10
```

aussi, encore une nouvelle fois, **sh**$<2>$ fourche-t'il en **sh**$<6>$ qui **execute****expr**.

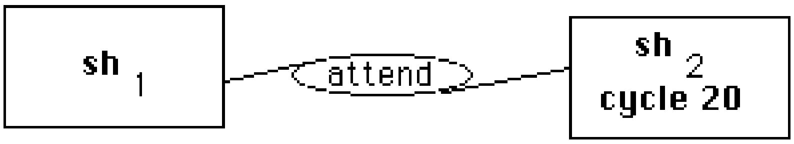


Ce dernier, après décodage de ses arguments, place 20 sur son flux sortant, récupéré par **sh**$<2>$ qui n'a plus à interpréter que

```
exec cycle 20
```



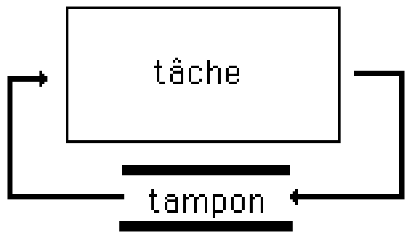
sh$<2>$ se transmue alors en **cycle** qui est un texte exécutable à interpréter, aussi se transmue-t'il à nouveau en **sh**, et tout recommence !!!



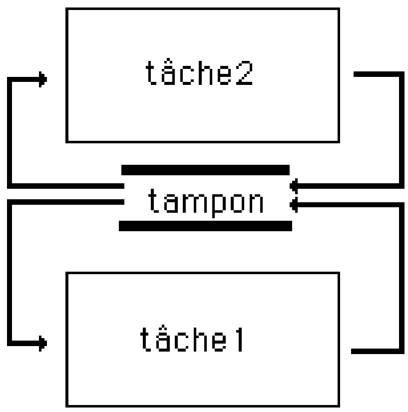
cycle permet tout bonnement d'exécuter un programme pendant des durées de plus en plus longues.

6.5 pipe

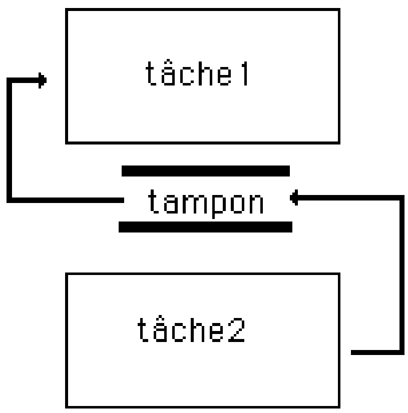
Cette dernière primitive que nous détaillerons, permet d'ouvrir des canaux de communications entre tâches. Elle ne se conçoit que grâce à **fork** et **exec** car un appel à **pipe** retourne deux flux, l'un en écriture et l'autre en lecture. Ces deux flux sont connectés par un tampon agissant comme une file d'attente : les caractères sont lus sur le canal selon l'ordre où ils y ont été écrits.



Une tâche seule n'a guère d'intérêt à appeler **pipe**. Toutefois après un **fork** on obtient



Et il suffit que la tâche mère n'utilise que son flux entrant tandis que la tâche fille n'use plus que de son flux sortant pour obtenir :



On a ainsi réalisé une cascade de deux tâches (et c'est ainsi que fait **sh** pour créer des schémas en cascade) où ecrivains et lecteurs sont synchronisés par le canal.

Annexe A

Bibliographie

P. Amar *Winnie manuel de Référence*, LRI Orsay, 1983
M.F. Banahan, A. Ruther *UNIX the book*, Sigma Technical Press, 1982
S.R. Bourne *The UNIX System*, Addison-Wesley, 1982
J. Chailloux *Le_Lisp de l'INRIA, Le Manuel de Référence*, 1985
K.Christian *The UNIX Operating System*, Wiley Interscience, John Wiley and Sons, 1983
? ? ?. Dax *Le langage C*, Eyrolles, 1984
? ? ?. Drappier, ? ? ?. Mauffrey *C par l'exemple*, Eyrolles, 1984
A.R. Feuer *The C Puzzle Book*, Prentice-Hall Software Series, 1978
B.W. Kernighan, R. Pike *The UNIX Programming Environnement*, Prentice-Hall Software Series, 1984
B.W. Kernighan, D.M. Ritchie *The C Programming Language*, Prentice-Hall Software Series, 1978
D.M. Ritchie, K. Thompson *The UNIX Time-Sharing System*, Bell System Technical Journal, Vol 57, #6, pp1905-1929, 1978
R.Thomas, J.Yates *A User Guide to the UNIX System*, Osborne/McGrawHill, 1982
Votre marchand d'ordinateur ... *UNIX Programmer's Manual, Vol 1*

Annexe B

L'alphabet ASCII

Octal	Décimal	Hexadécimal	Représentation	Code	Nom
0	0	0	^@	NUL	Null
1	1	1	^A	SOH	Start Of Heading
2	2	2	^B	STX	Start Of Text
3	3	3	^C	ETX	End Of Text
4	4	4	^D	EOT	End Of Transmission
5	5	5	^E	ENQ	Enquiry (Who are you ?)
6	6	6	^F	ACK	Acknowledge
7	7	7	^G	BEL	Bell
10	8	8	^H	BS	Backspace
11	9	9	^I	HT	Horizontal Tabulation
12	10	A	^J	LF	Line Feed
13	11	B	^K	VT	Vertical Tabulation
14	12	C	^L	FF	Form Feed
15	13	D	^M	CR	Carriage Return
16	14	E	^N	SO	Shift Out
17	15	F	^O	SI	Shift In
20	16	10	^P	DLE	Data Link Escape
21	17	11	^Q	DC1	Device Control 1
22	18	12	^R	DC2	Device Control 2
23	19	13	^S	DC3	Device Control 3
24	20	14	^T	DC4	Device Control 4
25	21	15	^U	NAK	Negative Acknowledge
26	22	16	^V	SYN	Synchronous idle
27	23	17	^W	ETB	End of Transmission Block
30	24	18	^X	CAN	Cancel previous data
31	25	19	^Y	EM	End of Medium
32	26	1A	^Z	SUB	Substitute character
33	27	1B	^[	ESC	Escape
34	28	1C	^\	FS	File Separator
35	29	1D	^]	GS	Group Separator
36	30	1E	^^	RS	Record Separator
37	31	1F	^_	US	Unit Separator
40	32	20		SP	Space
177	127	7F	^?	DEL	Delete

Octal	Décimal	Hexa	Lettre	Octal	Décimal	Hexa	Lettre
40	32	20	SP	120	80	50	P

41	33	21	!	121	81	51	Q
42	34	22	"	122	82	52	R
43	35	23	#	123	83	53	S
44	36	24	$	124	84	54	T
45	37	25	%	125	85	55	U
46	38	26	&	126	86	56	V
47	39	27	'	127	87	57	W
50	40	28	(	130	88	58	X
51	41	29	)	131	89	59	Y
52	42	2A	*	132	90	5A	Z
53	43	2B	+	133	91	5B	[
54	44	2C	,	134	92	5C	\
55	45	2D	-	135	93	5D	]
56	46	2E	.	136	94	5E	^
57	47	2F	/	137	95	5F	_
60	48	30	0	140	96	60	`
61	49	31	1	141	97	61	a
62	50	32	2	142	98	62	b
63	51	33	3	143	99	63	c
64	52	34	4	144	100	64	d
65	53	35	5	145	101	65	e
66	54	36	6	146	102	66	f
67	55	37	7	147	103	67	g
70	56	38	8	150	104	68	h
71	57	39	9	151	105	69	i
72	58	3A	:	152	106	6A	j
73	59	3B	;	153	107	6B	k
74	60	3C	<	154	108	6C	l
75	61	3D	=	155	109	6D	m
76	62	3E	>	156	110	6E	n
77	63	3F	?	157	111	6F	o
100	64	40	@	160	112	70	p
101	65	41	A	161	113	71	q
102	66	42	B	162	114	72	r
103	67	43	C	163	115	73	s
104	68	44	D	164	116	74	t
105	69	45	E	165	117	75	u
106	70	46	F	166	118	76	v
107	71	47	G	167	119	77	w
110	72	48	H	170	120	78	x
111	73	49	I	171	121	79	y
112	74	4A	J	172	122	7A	z
113	75	4B	K	173	123	7B	{
114	76	4C	L	174	124	7C	|
115	77	4D	M	175	125	7D	}
116	78	4E	N	176	126	7E	~
117	79	4F	O	177	127	7F	DEL