

Introduction à Unix

Nicolas Bedon

16 septembre 2010

Résumé

Ce document est une introduction à Unix pour un public sans connaissance particulière de l'informatique. Il demande toutefois de nombreuses explications orales. Après un bref aperçu de ce qu'est un ordinateur et un système d'exploitation, et Unix en particulier, l'interprète de commandes est introduit, et le shell est vu en détails. Des exercices sont proposés au fur et à mesure de la lecture.

Table des matières

1	Petite histoire de l'informatique	2
2	Structure physique d'un ordinateur	3
3	Le système d'exploitation	5
3.1	Le système de gestion de fichiers	5
3.2	La gestion de la mémoire	5
3.3	Les entrées/sorties	7
3.4	La gestion des tâches	7
3.5	L'interface homme-machine	8
3.5.1	Les applications	8
4	Unix	9
4.1	Le système de gestion de fichiers (SGF)	9
4.1.1	Les fichiers	9
4.1.2	Les répertoires	10
4.1.3	Les systèmes de fichiers	11
4.1.4	Les références	12
4.2	La gestion de la mémoire	12
4.3	Les entrées/sorties	13
4.4	La gestion des tâches	14
4.4.1	L'ordonnancement	14
4.4.2	La création	15

4.4.3	Les signaux	16
4.4.4	Le terminal de rattachement	17
4.4.5	Les redirections d'entrées/sorties standard	17
4.4.6	Les tubes	18
4.4.7	Les processus en tâche de fond	18
4.4.8	Le répertoire de travail	19
4.4.9	Les variables d'environnement	19
4.5	L'interface homme-machine	20
4.5.1	Interface graphique	20
4.5.2	Interprète de commandes	20
4.5.3	Les applications	20
5	Les principales commandes d'Unix	22
5.1	Commandes pour manipuler le système de fichiers	22
5.2	Commandes pour manipuler les processus	23
5.3	Commandes diverses	23
6	Le bash	24
6.1	bash : Introduction et rappels	24
6.2	Forme d'une expression régulière	41
6.3	Commandes utilisant les expressions régulières	45
6.4	Exercices	45
6.5	Programmation en bash	46
6.6	Scripts et paramètres	46
6.7	Structures de contrôle	48
6.8	Les fonctions	50
6.9	Exercices	51
6.10	Programmation avancée	51
6.11	Exercices	51
	Bibliographie	52
	Référence rapide d'Emacs	53

1 Petite histoire de l'informatique

L'informatique est en fait une très vieille idée, puisqu'elle vient de la mécanisation du travail.

Le tout premier ordinateur a été conçu dans la première moitié du *XIX^{ème}* siècle par le mathématicien anglais Babbage. Il n'a jamais fonctionné, les techniques mécaniques

employées étant trop précises pour l'époque. Sa réalisation a cependant été achevée récemment, et la machine en état de marche est aujourd'hui exposée au musée des sciences de Londres.

L'effort scientifique accompli durant la seconde guerre mondiale a permis la construction du premier ordinateur électrique, Colossus (UK), dont le rôle était de décrypter des messages secrets adverses. Vint ensuite, après la guerre, l'ENIAC, dont l'utilisation principale était le calcul de tables trigonométriques. Il était énorme, coûtait très cher, et de maintenance difficile : on le programmait électriquement, par câblage.

Cependant, les machines d'aujourd'hui fonctionnent sur les mêmes principes, dits de Von Neumann (un des principaux concepteurs de l'ENIAC) : simplement, le transistor a permis une meilleure intégration (donc une amélioration de la vitesse de traitement), une réduction des coûts et une meilleure fiabilité. La croissance (exponentielle¹) de la puissance des machines a permis d'étendre leur domaines d'application : Conception Assistée par Ordinateur (CAO), bases de données, documentation automatique, communication, aide à la décision. . . .

En théorie, tout ce qui est faisable par un ordinateur l'est aussi par procédé mécanique (des engrenages, par exemple), ce qui limite beaucoup l'ensemble des problèmes faisables par un ordinateur, qui est très petit par rapport à l'ensemble des questions qu'on peut se poser, et une question reste ouverte : l'ordinateur est-il intelligent (c'est-à-dire, l'intelligence est-elle un objet mécanique fini) ?

2 Structure physique d'un ordinateur

Un ordinateur n'est qu'une machine de traitement de données.

Ces données peuvent être de natures variées et l'utilisateur doit pouvoir les communiquer à l'ordinateur et lui donner les ordres qui correspondent aux traitements qu'il désire faire sur ces données. Enfin, l'ordinateur doit être capable de lui retourner les résultats qu'il a obtenus.

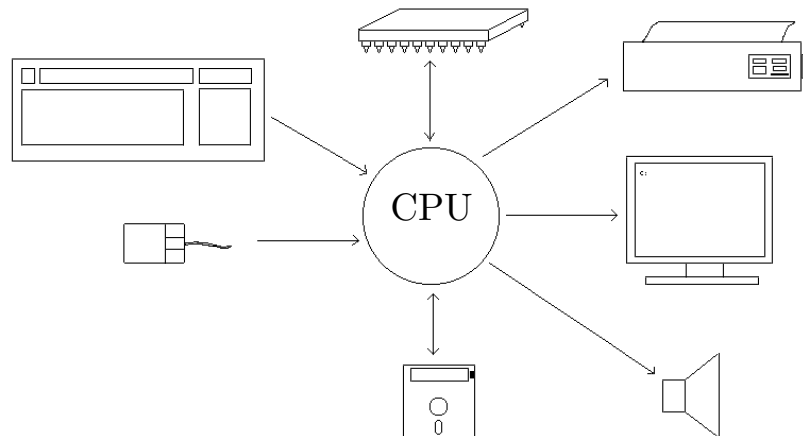
La structure même de l'ordinateur découle de cette idée simple. On passe en revue globalement les composantes d'un ordinateur :

- Le **clavier** est un moyen de communication avec la machine dans le sens homme vers machine. La **souris** en est un autre. Ce sont des **périphériques d'entrée**.
- L'**écran** est un des dispositifs permettant la communication dans le sens machine vers homme. C'est par cet appareil que la machine vous "parle". On dit que c'est un **périphérique de sortie** au même titre que l'imprimante, la table traçante, le haut-parleur, . . .
- Les **unités de stockage** – disque dur, bande magnétique, disquette, . . . – représentent l'équivalent d'une bibliothèque. Un nombre souvent impressionnant de données y est stocké. Comme la machine y écrit et y lit des informations, ce sont des **périphériques**

1. multipliée par 2 tout les 18 mois!!!

d'entrées-sorties. Mettre hors tension l'ordinateur n'affecte pas le contenu de ces unités de stockage que l'on appelle aussi **mémoires de masse** et qu'on divise en deux catégories : accès direct (disques durs, disquettes), et séquentiel (bandes).

- La **mémoire** de l'ordinateur, appelée **mémoire centrale** – RAM –, correspond plus à votre mémoire, celle dans laquelle votre cerveau extrait les informations dont il a besoin à un instant donné. Cependant une petite différence réside dans le fait que lorsque l'on éteint l'ordinateur, on perd à jamais le contenu de la mémoire centrale ! On observe une certaine similitude entre les mémoires RAM et les mémoires de masse : toutes servent à retenir de l'information, mais les deux classes de mémoire diffèrent par leur coût (la RAM est plus chère que la mémoire de masse, et donc en quantité plus petite sur les ordinateurs) et par leur rapidité (temps d'accès) : la lecture d'une information sur un disque dur est de l'ordre de 1 million de fois plus lente qu'en RAM.
- Et pour contrôler tout ça, le **processeur**. C'est le centre de la machine. A l'écoute des ordres qu'on lui donne *via* le clavier, il cherche ses instructions et les données à traiter dans la mémoire, exécute les instructions sur les données et peut par exemple ordonner l'affichage des résultats sur l'écran, ou simplement stocker les résultats en mémoire centrale ou en mémoire de masse.
- Pour que tout ces éléments communiquent les uns avec les autres – par exemple lorsque le processeur va chercher une information dans la mémoire centrale –, ils sont reliés par des connexions électriques que l'on appelle des **bus** dans lesquelles circulent les informations. Une **horloge interne** synchronise le tout.



3 Le système d'exploitation

Il masque à l'utilisateur la vérité à propos de la structure matérielle de la machine, en tentant de rendre chaque entité qui compose un ordinateur facile à utiliser. Par exemple, lorsque l'utilisateur souhaite accéder à un fichier sur disquette, le système d'exploitation se charge de la faire tourner à la bonne vitesse, d'y localiser les données, de positionner les têtes de lecture, d'amener les données en mémoire centrale et finalement d'arrêter le moteur du lecteur de disquettes.

Techniquement, on le divise en cinq parties essentielles :

- Gestion des tâches en cours d'exécution.
- Gestion des entrées/sorties.
- Gestion de la mémoire.
- Système de fichiers.
- Interface homme↔machine.

Le système d'exploitation se charge donc de toutes ces opérations, sans que l'utilisateur ait à savoir ce qui se passe réellement dans la machine ; ainsi les utilisateurs ne connaissant rien à sa véritable structure peuvent quand même l'exploiter de façon convenable.

Les systèmes d'exploitation modernes sont :

- Multi-utilisateurs, c'est-à-dire qu'ils permettent à plusieurs utilisateurs d'utiliser une machine en même temps, le système se chargeant d'allouer un temps de calcul à chaque utilisateur de manière équitable
- Multi-tâches, c'est-à-dire qu'ils sont capables d'exécuter plusieurs tâches en même temps, ou du moins de le faire croire.

3.1 Le système de gestion de fichiers

Sur les unités de sauvegarde – disque dur, disquette,..–, les données sont stockées sous forme d'entités séparées les unes des autres et repérables par une **référence** : ce sont les **fichiers**. La partie du système d'exploitation qui gère les fichiers est le système de fichiers. Le système de fichiers utilisé par un système d'exploitation dépend essentiellement de l'utilisation principale de l'ordinateur. Ainsi, un ordinateur dédié aux bases de données aura un système de fichiers adapté, afin d'améliorer les performances des opérations faites par les bases de données sur les fichiers. Un système de fichiers spécialisé peut se révéler inefficace quand on sort de son domaine de spécialisation. Ainsi, la plupart des ordinateurs n'étant pas dédiés à une tâche particulière, leur système de fichiers n'est pas spécialisé.

3.2 La gestion de la mémoire

Globalement, une mémoire est un endroit où stocker des informations. Comme vu précédemment, il existe sur les ordinateurs deux gros types de mémoire :

- la mémoire basée sur des composants uniquement électronique, qui est chère, mais rapide, et qui présente le désavantage de nécessiter une alimentation électrique constante
- la mémoire basée sur des composants électroniques et mécaniques, qui est peu coûteuse, mais lente (de l'ordre de un million de fois plus lente que la précédente, la lenteur étant due aux composantes mécaniques), mais où l'information est persistante à une non alimentation électrique.

Le rôle du gestionnaire de mémoire est d'organiser les données dans l'ensemble de la mémoire pour le système d'exploitation. Le gestionnaire de mémoire est invisible à l'utilisateur, contrairement au gestionnaire de fichiers.

Bien entendu, le gestionnaire de mémoire doit faire son travail de la manière la plus efficace possible. Ainsi, il doit placer judicieusement les données, en fonction des types de mémoire dont il dispose et des caractéristiques des données à placer. Par exemple, il essaie de placer les données fréquemment utilisées dans la mémoire la plus rapide.

Il différencie également les zones de mémoire utilisées des zones inutilisées au cours du temps. Le nombre de zones et leurs tailles varient au cours du temps. Les données mises dans une zone de mémoire ne pouvant être déplacées, le travail du gestionnaire de mémoire est difficile : il doit gérer la mémoire de manière à en économiser le plus possible, mais le gestionnaire étant très utilisé, il doit également être le plus rapide possible. Cette double contrainte d'économie à la fois de temps et d'espace fait partie des problèmes difficiles de l'informatique. En informatique, les économies de temps se traduisent souvent par une gestion moins efficace de la mémoire, et inversement, faire des économies d'espace nécessite souvent d'avoir recours à des calculs plus compliqués (comme la compression des données), donc une perte de temps. Il faut donc faire un compromis entre perte de temps et perte d'espace ; ce compromis est souvent décidé sur des critères expérimentaux.

Le gestionnaire de mémoire peut fournir également au programmeur **un espace d'adressage virtuel** de la mémoire. Ainsi, chaque programme manipule la mémoire à son niveau d'abstraction propre, sans avoir à se préoccuper de la manière dont est réellement (physiquement) exploitée la mémoire. Par exemple, sur un système multi-tâche, où cohabitent simultanément plusieurs programmes, qui ont donc à se partager la mémoire, chaque programme voit la mémoire comme s'il était le seul à l'utiliser, ce qui évidemment n'est pas le cas. Ce procédé permet en particulier de garantir qu'un programme ne pourra pas accéder aux données d'un autre programme : outre le fait que l'adressage virtuel facilite la tâche du programmeur, il offre également des garanties de sécurité.

Selon les systèmes d'exploitation utilisant l'adressage virtuel, l'espace d'adressage virtuel d'un programme est divisée en **pages**, i.e. en morceaux de taille constante, ou en **segments**, i.e. en morceaux dont la taille peut varier, ou bien encore en combinant les techniques de pagination et de segmentation. Ce découpage permet au gestionnaire de mémoire de placer les morceaux, pages ou segments, de mémoire virtuelle dans les mémoires physiques en fonction de la fréquence d'accès des données contenues dans la page ou au segment et de la vitesse d'accès des différents types de mémoire disponibles.

Le rôle du gestionnaire de mémoire est primordial, et très compliqué : la mémoire fait en effet partie des ressources critiques des systèmes informatiques. S'il gère mal les espaces

mémoires disponibles ou occupés, en considérant occupées des zones de mémoires qui ne sont jamais utilisées, certaines applications vont manquer de mémoire. Si au contraire il les gère trop finement, cette gestion sera coûteuse en temps et les programmes s'exécuteront trop lentement. Une erreur de gestion de l'adressage virtuel se traduira par des problèmes de cohérences des données, et de sécurité. Mettre des données fréquemment utilisées dans des mémoires lentes, comme par exemple sur un disque dur ou une disquette, fera chuter dramatiquement les performances du système, le rendant inutilisable (et comment décider à l'avance quelles seront les données fréquemment utilisées?).

3.3 Les entrées/sorties

L'ensemble processeur+bus+mémoire a la capacité de faire des calculs complexes. Il faut cependant lui indiquer comment faire ces calculs, et sur quoi, et il faut être capable de récupérer le résultat du calcul. En d'autres termes, il faut être capable de communiquer avec lui. C'est le rôle de la partie du système d'exploitation désignée par le terme générique "entrées/sorties". C'est cette partie qui s'occupe de la gestion des périphériques, par l'intermédiaire de modules appelé "drivers". Par exemple, elle s'occupe de traiter ce que tape l'utilisateur au clavier, de l'affichage sur l'écran, de l'envoi de données à l'imprimante, de la lecture de données sur un disque,.... Cette partie très compliquée du système est divisée en couches, qui permettent de distinguer les entrées/sorties logiques (où on fait abstraction des caractéristiques physique des périphériques) des entrées/sorties physiques (où on manipule les caractéristiques physiques des périphériques). On peut donc faire des entrées/sorties avec plusieurs niveaux d'abstraction, mais sur les systèmes multi-utilisateurs, seuls les niveaux d'abstraction les plus hauts sont accessibles aux utilisateurs (pour des raisons évidentes de sécurité), les niveaux d'abstraction les plus faibles étant exclusivement réservés à l'usage du système lui-même. Par exemple, l'utilisateur peut demander au système de lire le premier mot du fichier de nom `toto` qui se trouve sur un cd, mais il ne peut pas lui demander de lire les 32 bits se trouvant à trois centimètres du centre du disque, à l'angle 90 par rapport à un axe de repère sur le disque, bien que le système sache le faire.

D'une manière générale, les entrées/sorties représentent une part importante du code écrit pour un système d'exploitation. En effet chaque périphérique a ses propres caractéristiques physiques, et un fonctionnement propre, que le système doit connaître de manière détaillée pour pouvoir exploiter pleinement le périphérique.

3.4 La gestion des tâches

Un "programme" est une tâche que l'ordinateur doit accomplir. Ces programmes sont une suite d'instructions élémentaires à exécuter (par le processeur) sur des données. Les processeurs ont une telle puissance de traitement (les instructions élémentaires sont exécutées tellement vite) qu'ils peuvent faire croire à l'utilisateur, en exécutant pendant quelques centièmes de seconde des instructions d'une première tâche, puis pendant quelques centièmes des instructions d'une seconde tâche, puis en revenant à la première, etc, que plusieurs

tâches (deux dans notre exemple) sont simultanément en cours d'exécution sur l'ordinateur. Les systèmes d'exploitation ainsi capables de répartir la puissance de traitement du processeur entre différentes tâches sont dits multi-tâches, les autres sont mono-tâche. La très grande majorité des systèmes d'exploitation installés sur les ordinateurs courants sont multi-tâches. Le "gestionnaire de tâche" est la partie du système qui s'occupe de la gestion des tâches. La capacité d'un système à traiter plusieurs tâches ouvre la voie à la multi-utilisation du système. De tels systèmes, qui sont donc utilisables par plusieurs utilisateurs simultanément, sont dits "multi-utilisateurs". De nombreux systèmes sont multi-utilisateurs.

3.5 L'interface homme-machine

Le système d'exploitation est donc un programme capable de gérer les différentes composantes matérielles d'un ordinateur. Il sait communiquer avec elles, et donc peut les faire communiquer entre elles. Reste maintenant à faire communiquer l'homme avec le système. C'est le rôle de l'interface homme-machine (IHM). On la trouve communément sous deux formes :

- une interface graphique, par laquelle le principal moyen d'expression de l'utilisateur est un périphérique de pointage, comme la souris. Ce moyen de communication est intuitif, il ne nécessite donc pas de connaissance particulière pour pouvoir être utilisé. Il est par contre très limité, et donc de faible puissance d'expression. D'autre part, tout traitement graphique étant coûteux en terme de temps de calcul, les interfaces graphiques nécessitent des machines suffisamment puissantes.
- une interface qui permet à l'utilisateur de communiquer avec la machine par l'intermédiaire du clavier, en employant un vocabulaire et une syntaxe simple dans un cadre rigide. Cette interface est appelée "interprète de commandes", parce qu'elle traduit les ordres tapés au clavier par l'utilisateur vers le système d'exploitation. Ce moyen de communication est plus puissant qu'une interface graphique, par contre il nécessite des connaissances du langage et de la syntaxe. Il est économique en terme de temps de calcul : les ordinateurs peu puissants sont donc équipés de ce type d'interface homme-machine.

3.5.1 Les applications

Ce sont des programmes complexes tels que

- tableur,
- traitement de textes, d'images, de sons,
- jeu,
- gestionnaire de courrier électronique
- ...

4 Unix

C'est un système d'exploitation né du milieu universitaire dans les années 60. Il est multi-tâches et multi-utilisateurs. Il était à l'origine essentiellement déployé sur de gros calculateurs. Les ordinateurs étant aujourd'hui très puissants, on le trouve maintenant sur tout type de machines, par exemple sur les PCs. Il en existe plusieurs versions, certaines étant commerciales, et d'autres gratuites, toutes offrant les mêmes fonctionnalités, avec cependant de petites incompatibilités. Il est très répandu grâce à sa version produite par GNU (www.gnu.org) et Linus Torvalds, nommée GNU-Linux, dont on peut trouver différentes distributions (RedHat, Mandrake, Debian, . . .) à petit prix sur cd et même gratuitement par Internet. C'est de plus un système d'exploitation étudié et utilisé à l'université. Il a part contre le désavantage d'être d'installation, de configuration et d'administration beaucoup plus difficile que des systèmes d'exploitation moins puissants comme Windows, mais cet écart tend à se réduire, les deux administrations tendant à se rapprocher.

Unix étant un système multi-utilisateurs, tout utilisateur doit s'identifier pour pouvoir s'en servir pendant une "session de travail". Les utilisateurs s'identifient par un couple nom d'utilisateur (aussi appelé nom de login), mot de passe. Le mot de passe est en général choisi par l'utilisateur, qui doit le tenir secret. La session d'utilisation commence juste après identification. Quand l'utilisateur ne souhaite plus se servir de la machine, il doit explicitement terminer (ou fermer) sa session de travail, afin qu'un autre utilisateur mal intentionné ne puisse pas utiliser son identité pour se servir de l'ordinateur. Durant sa session de travail, l'utilisateur est repéré par le système grâce à un User Identifier (UID), numéro associé à l'identification de l'utilisateur. Chaque utilisateur fait partie d'un ou plusieurs groupes définis par l'administrateur du système. Chaque groupe est repéré par le système grâce à un Group Identifier (GID).

Nous allons maintenant passer en revue les principales caractéristiques d'Unix.

4.1 Le système de gestion de fichiers (SGF)

4.1.1 Les fichiers

Les fichiers sous Unix sont non typés, c'est-à-dire que le SGF ne stocke pas d'information sur la nature des informations qu'ils contiennent. Tout type d'information peut être mis dans un fichier. Les applications qui respectivement écrivent et lisent les données doivent donc être d'accord sur le typage des données. Les fichiers peuvent être de taille quelconque (en théorie bien entendu, en pratique le nombre représentant la taille maximale est si grand qu'on a du mal à lui associer une signification). Le SGF conserve cependant des informations sur chaque fichier :

- sa taille (en octets)
- l'UID de son créateur. Le créateur du fichier est aussi appelé son "propriétaire".
- le GID de son propriétaire
- sa date de création
- sa date de dernière modification

- sa date de dernier accès
- son nom
- un entier qui identifie le fichier sur le système de fichiers (nous en parlerons plus tard), appelé “numéro d’inode” ou en français “numéro d’i-nœud”. à un numéro d’inode ne correspond qu’un fichier
- ses droits d’accès

Les droits d’accès à un fichier sont fixés par son propriétaire au moment de sa création. Ce dernier peut à tout moment les changer. Les droits d’accès peuvent interdire à un autre utilisateur du système d’aller lire ou écrire dans le fichier. De manière plus détaillée, les droits d’un fichier s’expriment en trois parties :

- les droits du propriétaire
- les droits des utilisateurs du même groupe que le propriétaire
- les droits des autres utilisateurs

Pour chacune des trois parties, on peut spécifier trois droits :

- le droit en lecture (Read permission)
- le droit en écriture (Write permission)
- le droit en exécution (eXecute permission). En effet, les programmes sont eux mêmes stockés dans des fichiers. On peut spécifier que le fichier contient un programme en autorisant l’exécution. Les fichiers ne contenant pas de programme ne devraient jamais avoir de droit en exécution, cependant on peut le mettre.

Notons que malheureusement ce système de droits n’est pas assez discriminant. En effet, si un utilisateur veut donner le droit d’accès à un fichier à un autre utilisateur, et seulement un seul, il est obligé de demander à l’administrateur du système de créer un groupe qui ne contiendra que l’autre utilisateur. L’administrateur du système n’ayant pas que ça à faire, il refusera.

Les noms sont des suites de caractères quelconques, et de taille limitée à moins d’1Ko sur la plupart des Unix modernes, 14 octets sur les vieilles versions d’Unix. Par soucis de compatibilité, on se restreint moralement à 14 caractères. Ils permettent de retrouver le fichier : à un nom complet (nous verrons ce que cela signifie plus tard) de fichier ne correspond qu’un fichier. Par contre, un fichier peut avoir plusieurs noms (il en a toujours au moins un).

Les opérations utilisateur permises sur les fichiers sont :

- y lire quand on en a le droit
- y écrire (c’est-à-dire le modifier) quand on en a le droit
- l’“exécuter”, quand on en a le droit
- le transfert de propriété, quand on en est propriétaire

4.1.2 Les répertoires

Un disque dur moyen contient actuellement plusieurs centaines de milliers de fichiers. Afin de pouvoir s’y retrouver, il est nécessaire de pouvoir les classer. Pour permettre ce

classement, Unix met à la disposition des utilisateurs des répertoires, qu'on peut assimiler à des boîtes dans lesquelles on peut mettre des fichiers et d'autres répertoires. Un répertoire doit être mis dans un répertoire, et dans un seul (contrairement aux fichiers : ces derniers pouvant avoir plusieurs noms, on peut les placer dans plusieurs répertoires). Cela signifie que l'ensemble des fichiers et répertoires est organisé suivant une structure hiérarchique arborescente, et que le tout premier répertoire a un statut spécial : en effet, puisqu'il doit par convention être contenu dans un répertoire, on dira par convention qu'il est contenu dans lui-même. Cette arbre est la plupart du temps assimilé à un arbre généalogique, on parle de répertoires pères et fils. Le répertoire de hiérarchie la plus haute est dit "racine". Unix gère les répertoires presque comme des fichiers : ils ont les mêmes attributs, avec une information supplémentaire qui permet de différencier les vrais fichiers des répertoires. Cette gestion et cette information permettent à Unix de voir les répertoires comme des fichiers, mais d'interdire à l'utilisateur de le faire.

Les informations conservées par Unix sur les répertoires sont les mêmes que pour les fichiers, avec un léger changement de sémantique au niveau des droits :

- avoir le droit en lecture sur un répertoire signifie avoir le droit d'obtenir le nom des objets qu'il contient
- avoir le droit en écriture sur un répertoire signifie avoir le droit d'y ajouter, de supprimer ou de changer le nom d'un élément dont on connaît le nom
- avoir le droit en exécution sur un répertoire signifie avoir le droit d'accéder à un élément du répertoire dont on connaît le nom

Les opérations utilisateur permises dans un répertoire sont, à condition d'en avoir le droit :

- y créer un répertoire ou un fichier
- y supprimer un répertoire ou un fichier
- y changer de nom un répertoire ou un fichier
- en transférer la propriété

Sous Unix les répertoires ne sont jamais totalement vides. Ils contiennent toujours au moins deux entrées. L'entrée de nom `..` désigne le répertoire père. L'entrée de nom `.` désigne le répertoire lui-même (tout répertoire contient donc toujours une référence à son père, et à lui-même). Par abus de langage nous dirons dans la suite qu'un répertoire est vide quand il ne contient que les entrées `.` et `..`.

4.1.3 Les systèmes de fichiers

Un système de fichiers Unix est un arbre de répertoires et fichiers dans lequel chaque élément est repéré par son numéro d'inode. Il est dans la plupart des cas associé à une mémoire de masse physique, et permet de voir logiquement cette mémoire de masse physique. Unix permet d'assembler plusieurs systèmes de fichiers dans une même arborescence : construire un arbre à partir de plusieurs, deux par exemple, en greffant le premier dans un répertoire vide du second. Cette opération s'appelle le "montage". Elle est très courante, car très pratique : en effet elle permet d'assembler plusieurs mémoires de masse physiques

et de les voir comme une seule unité logique. Par contre, il présente un inconvénient : un numéro d'inode peut être associé à plusieurs fichiers, puisqu'il peut être associé à un fichier dans plus d'un système de fichiers.

4.1.4 Les références

Il existe deux moyens de référencer une entrée d'un système de fichiers : les références relatives et les références absolues. Les références non triviales sont aussi appelées "chemins".

Les références absolues

Elles déterminent comment accéder à l'entrée en partant de la racine de l'arbre du SGF. Elles commencent par un /, et le séparateur entre les composantes est également le caractère /. Supposons par exemple que la racine du SGF contiennent un répertoire `usr`, qui lui même contient un répertoire `bin`, qui lui contient une entrée de nom `man`. une référence absolue à cette entrée est `/usr/bin/man`. Une autre serait `/usr/./usr/bin/./man`. Notons qu'on peut en construire une infinité, qui sont toutes équivalentes.

Les références relatives

Elles sont décrites plus loin, quand la notion de "répertoire de travail" d'un processus est introduite.

4.2 La gestion de la mémoire

Sous Unix, le gestionnaire de mémoire permet d'offrir aux programmes la possibilité d'utiliser de manière transparente, et assez efficace, les différents types de mémoire disponibles de manière transparente. Par exemple, si une machine possède x kilo-octets de mémoire électronique, et y kilo-octets de mémoire de masse, le programme disposera de $x + y$ kilo-octets de mémoire, sans savoir qu'ils sont répartis en x kilo-octets de mémoire électronique, et y kilo-octets de mémoire de masse. Les y kilo-octets sur la mémoire de masse auront, au moment de l'installation d'Unix, été déclarés comme étant utilisables comme de la mémoire (et non utilisables pour y mettre un système de fichiers). On dit qu'ils forment une **partition de swap**.

Chaque programme dispose également sous Unix d'un espace d'adressage virtuel (une des raisons étant qu'un tel système d'adressage permet de régler de manière élégante les problèmes de sécurité liés à la coexistence de plusieurs programme au même instant sur le système).

Sous Unix, la mémoire est également paginée et segmentée. De plus, différents programmes ont la possibilité de partager un espace mémoire commun : c'est la **mémoire partagée**. La mémoire paginée permet d'économiser de l'espace mémoire, car il évite d'avoir des données dupliquées. Elle fournit également un moyen de communication entre programmes.

4.3 Les entrées/sorties

Les entrées-sorties sous Unix, comme sous tout système d'exploitation, font appel à des mécanismes complexes, et sont la plupart du temps très lentes : en effet par définition elles vont actionner un périphérique, qui bien souvent comporte des parties mécaniques. De plus les mécanismes compliqués mis en jeu sont lents. Même l'utilisation de périphériques entièrement électroniques, comme l'écran par exemple, est très lente. Un programmeur averti les évitera donc au maximum. Ainsi, un programme qui fait des calculs intensifs évitera au maximum d'écrire des résultats sur l'écran, dans un fichier ou sur une imprimante. En plus de cette lenteur, les entrées-sorties soulèvent un autre problème sur les systèmes multi-tâches comme Unix : l'accès exclusif à la ressource. Quand une tâche imprime, par exemple, il n'est pas question qu'une autre tâche imprime en même temps. Il faut donc que la première réserve la ressource, et que la seconde attende que la première ait fini avant de commencer. Le système d'exploitation a donc un rôle d'arbitrage en ce qui concerne l'accès aux ressources. D'autre part, pour des raisons de sécurité, tout accès aux ressources doit se faire par l'intermédiaire du système d'exploitation.

Unix partitionne la classe des périphériques en deux :

- les périphériques fonctionnant en mode “bloc”, pour lesquels les données sont directement accessibles, et qui utilisent le mécanisme de bufferisation, expliqué un peu plus loin. L'exemple le plus courant de périphérique en mode bloc est le disque magnétique ou optique. L'unité de lecture/écriture d'information sur ce type de périphérique est le bloc, groupe d'octets (par exemple 512), qu'on peut transférer directement de la mémoire vers le support physique. Chaque bloc est repéré par un numéro, et est accessible indépendamment des autres.
- les périphériques fonctionnant en mode “caractère”, qui sont vus comme des flots de caractères, et qui ne supportent pas la bufferisation. Un exemple typique de ce genre de périphérique est la bande magnétique. Pour lire le 500ème caractère sur la bande, il faut lire un par un les 499 précédents.

Cette partition est une approximation. En effet, théoriquement, certains périphériques ne rentrent dans aucune des classes. Dans laquelle, par exemple, placer une horloge ? Cependant, la très grande majorité des périphériques rentre dans une de ces deux classes.

Un autre rôle du système est de fournir des accès optimisés (les plus rapides possibles) aux périphériques. Pour assurer cette fonction, Unix utilise des techniques de **bufferisation** (ou **cache**) des lectures/écritures pour les périphériques sur lesquels cette technique est pertinente. Supposons par exemple qu'un programme veuille écrire “toto” dans un fichier sur une disquette. Pour cela, le programmeur malhabile fait quatre appels au système : un pour écrire chaque lettre, dans l'ordre. Le temps d'écriture sur une disquette, qui est de l'ordre d'un dixième de seconde, est réparti comme suit : les huit dixièmes pour démarrer le moteur qui fait tourner la disquette, et attendre qu'il tourne à la bonne vitesse, un dixième pour déplacer la tête d'écriture au bon endroit et la descendre sur le disque, et le dernier dixième pour écrire le caractère et remonter la tête. Les quatre écritures prennent donc quatre dixièmes de seconde si les écritures se font de manière synchrones avec les demandes d'écritures. Pour gagner du temps, le système procède de manière plus intelligente : il

mémorise les zones récemment accédées au disque en les plaçant en mémoire, et écrit les modifications apportées à cette zone dans la mémoire plutôt que sur le disque. Quand le système en juge le moment opportun, les données sont réellement écrites sur le disque. On évite ainsi des activations inutiles de parties mécaniques. Le système fait attention à ne pas générer d'informations incohérentes entre les données qu'il a en mémoire et les données réellement présentes sur le disque.

Les fonctions d'entrées/sorties fournies par Unix sont assez sommaires, et d'utilisation complexe. Afin de faciliter les entrées/sorties, la plupart des langages de programmation (même le C) fournissent au programmeur des bibliothèques d'entrées/sorties, qui ont elles-mêmes un niveau de bufferisation propre.

D'autre part, Unix met à la disposition des utilisateurs des commandes permettant l'utilisation des périphériques, comme par exemple des commandes d'impression, de formatage de disquette, ...

4.4 La gestion des tâches

Sous Unix, un programme en cours d'exécution s'appelle un *processus*. Chaque processus a son propre espace d'adressage virtuel. De plus, chaque processus possède les caractéristiques suivantes :

- un numéro, fixé par le système, qui l'identifie
- un groupe,
- un propriétaire, qui est l'utilisateur qui exécute le programme,
- un nom d'exécutable,
- un processus créateur, dit "père"
- une taille
- une priorité, qui peut être modifiée par le propriétaire
- une fréquence d'exécution, calculée par l'ordonnanceur
- un état d'exécution, calculé par l'ordonnanceur
- une table des signaux à traiter, fournie par le système, et dans laquelle le propriétaire peut ajouter des signaux à traiter
- une table des fichiers ouverts
- un terminal de rattachement, à partir duquel le processus peut récupérer des signaux à traiter
- un répertoire de travail
- des variables d'environnement
- un environnement d'exécution, fourni par le système
- tout un tas d'autres choses qui sont expliquées dans les livres. . .

4.4.1 L'ordonnement

Unix est un système multi-tâches. Plusieurs tâches peuvent donc être "simultanément" en cours d'exécution. Pourtant, dans la plupart des cas, l'ordinateur ne dispose que d'un processeur. Le rôle de l'**ordonnanceur de processus** est de découper le temps de calcul

du processeur entre les processus, en étant le plus équitable possible, en faisant en sorte que l'exécution de chaque processus soit la plus rapide possible, en fonction des propriétés des processus. Par exemple, un processus qui attend qu'un autre processus libère une ressource n'a pas besoin d'être exécuté (puisque l'exécuter revient à attendre la libération de la ressource). Plutôt que de le laisser dans un état d'attente active (qui consomme de la ressource CPU), l'ordonnanceur va exécuter les autres processus jusqu'à ce que la ressource se libère. Les processus sous Unix ont globalement trois états :

- en cours d'exécution : le processeur lui est attribué
- prêt : le processus est prêt à être exécuté, il attend seulement que le processeur lui soit attribué par l'ordonnanceur
- en attente d'évènement : l'ordonnanceur attend qu'un évènement, comme par exemple la libération d'une ressource, survienne pour que reprenne l'exécution du processus

La mise à disposition du processeur à un processus par l'ordonnanceur est faite en fonction de :

- son état
- sa priorité
- sa fréquence d'exécution

Une propriété importante de l'ordonnanceur de processus d'Unix est que comme il essaie d'être équitable, les applications temps réel sont interdites : en effet les applications temps réel doivent pouvoir demander que le processeur leur soit allouer pendant un temps donné, une seconde par exemple, ce que l'ordonnanceur Unix ne peut pas garantir. Si c'était le cas il ne serait pas équitable vis-à-vis des autres processus.

4.4.2 La création

Il n'existe sous Unix qu'un moyen de créer un processus : la duplication d'un processus existant (exception faite du tout premier processus, dont le rôle est spécial et qui est créé de manière spéciale). La duplication doit se faire à partir d'un processus. Un processus ne peut pas dupliquer un autre processus : il doit se dupliquer lui-même s'il veut créer un autre processus. La création d'un processus correspondant à l'exécution d'un programme particulier se fait en deux étapes : la duplication d'un processus, et le recouvrement du duplicata par le code et les données du programme à exécuter. Supposons par exemple qu'on veuille exécuter un programme de nom p à partir d'un interprète de commandes. Les instructions et données nécessaires à l'exécution de ce programme se trouvent dans le fichier de nom p . Pour exécuter p , il suffit de taper p dans l'interprète de commande. Une fois l'analyse de ce qu'a tapé l'utilisateur terminée, l'interprète de commande va créer un autre processus en se dupliquant. Le processus nouvellement créé (appelé "fils") est un duplicata presque exact de son original (son "père"). Les seules différences sont que les numéros de processus ne sont pas identiques, et qu'une donnée permet aux codes des fils et du père de s'auto-différencier. Ainsi, le fils sait qu'il est fils et le père sait qu'il vient de créer un fils. Le fils est programmé de telle manière qu'il doive se recouvrir par le contenu du fichier p . Une fois le fils recouvert par le contenu du fichier p , l'exécution du programme

correspondant peut commencer. Le père est programmé de telle manière qu'il attende que le fils se termine (on emploie l'horrible terme de "mort" du fils), sauf mention du contraire.

4.4.3 Les signaux

Unix a, entre autres, un moyen très primaire de communiquer avec les processus, qu'il met à la disposition des processus pour communiquer entre eux : ce sont les signaux. On "envoie" un signal à destination d'un processus. Un signal est identifié par un petit entier, auquel on associe un nom symbolique pour des raisons mnémotechniques. Chaque signal a une signification. Par exemple, l'envoi du signal SIGKILL à un processus signifie qu'on veut le tuer, c'est-à-dire qu'il se termine immédiatement. Certains signaux peuvent être attrapés et gérés par le processus destinataire. Ces signaux peuvent alors être vus comme des interruptions. D'autre, par contre, comme SIGKILL, ne le sont pas. Ainsi, un processus qui reçoit un SIGKILL va être tué par le système sans qu'il ait conscience de sa mort, puisqu'il ne peut ni attrapé, ni gérer le signal. Certains signaux sont à l'usage de l'utilisateur d'Unix, qui s'en sert alors comme moyen de communication avec le processus. Leur sémantique n'est pas définie par Unix (exemple : SIGUSR1). D'autres, par contre, sont à l'usage du système : ils permettent au système d'indiquer au processus qu'il a exécuté une instruction ayant abouti à une erreur (adressage illégal : SIGSEGV, erreur mathématique en virgule flottante : SIGFPE;...). Dans ce cas des actions sont associés au signal.

Voici les signaux les plus couramment utilisés :

Signal	Valeur	Action	Commentaire
SIGHUP	1	A	Raccrochement (déconnexion) sur terminal de contrôle, ou mort du processus de contrôle.
SIGINT	2	A	Interruption depuis le clavier.
SIGQUIT	3	A	Demande 'Quitter' depuis le clavier.
SIGILL	4	A	Instruction illégale.
SIGABRT	6	C	Signal d'arrêt depuis abort(3).
SIGFPE	8	C	Erreur mathématique virgule flottante.
SIGKILL	9	AEF	Signal 'KILL'.
SIGSEGV	11	C	Référence mémoire invalide.
SIGPIPE	13	A	Écriture dans un tube sans lecteur.
SIGALRM	14	A	Temporisation alarm(2) écoulée.
SIGTERM	15	A	Signal de fin.
SIGUSR1	30,10,16	A	Signal utilisateur 1.
SIGUSR2	31,12,17	A	Signal utilisateur 2.
SIGCHLD	20,17,18	B	Fils arrêté ou terminé.
SIGCONT	19,18,25		Continuer si arrêté.
SIGSTOP	17,19,23	DEF	Arrêt du processus.

et les actions associées :

- A Par défaut, terminer le processus.
- B Par défaut, ignorer le signal.

- C Par défaut, créer un fichier core.
- D Par défaut arrêter le processus.
- E Le signal ne peut pas être intercepté.
- F Le signal ne peut pas être ignoré.

4.4.4 Le terminal de rattachement

L'idée d'associer un terminal de rattachement à un processus vient de l'observation suivante : un utilisateur ouvre une session de travail. À partir de son terminal, il lance un programme. Puis, il ferme sa session de travail en déconnectant son terminal. Que devient le programme ? Doit-on continuer son exécution ou non ? Sous Unix, à part mention d'un comportement explicite différent, le programme reçoit un signal (SIGHUP), qui lui indique que le terminal à partir duquel il a été lancé est déconnecté, et donc qu'il doit mourir. Le terminal de rattachement a un autre rôle. L'interprète de commandes lit les commandes tapées par l'utilisateur au clavier, et affiche ses sorties sur l'écran. Le clavier est l'**entrée standard** du processus interprète de commandes. L'écran est sa **sortie standard**. Chaque processus possède trois entrées/sorties standard, qui peuvent être n'importe quel fichier ou périphérique (bien entendu, sur lesquels les opérations sont permises : on n'attendra pas d'une imprimante qu'elle soit un périphérique d'entrée de données) :

- l'entrée standard, à partir de laquelle on attend les données, sauf mention d'une autre entrée
- la sortie standard, sur laquelle on met les messages de sortie, sauf mention d'une autre sortie
- la sortie erreur standard, sur laquelle on met les messages d'erreurs, sauf mention d'une autre sortie d'erreur

Ces entrées/sorties standard sont transmises du processus père au fils au moment de la duplication. Ainsi, lorsque l'interprète de commandes crée le processus fils qui va être recouvert par la commande à exécuter, cette dernière récupère les entrées/sorties standard de l'interprète de commande. Elle lira donc entrées sur le clavier, sauf mention du contraire et affichera ses messages à l'écran, sauf mention du contraire. Ceci explique que l'interprète de commande attende la mort de la commande lancée avant de reprendre son travail : en effet, si ça n'était pas le cas, et que l'utilisateur tapait quelque chose au clavier, à qui serait destiné ce que l'utilisateur tape ? à l'interprète de commandes ou au programme qu'il vient de lancer ? Les entrées/sorties standard d'un interprète de commandes sont bien entendu par défaut toujours le clavier et l'écran. Le terminal de rattachement est, comme les autres propriétés des processus, hérité de père en fils. lorsqu'un utilisateur déconnecte son terminal, l'interprète de commande associé au terminal et tout ses fils sont donc informés de la déconnection et se terminent, sauf mention contraire.

4.4.5 Les redirections d'entrées/sorties standard

Les entrées/sorties standard sont héritées de processus père en fils. Bien entendu, sous Unix, un processus peut à tout moment décider de changer ses entrées/sorties standard

vers d'autres périphériques, temporairement ou définitivement. Ce mécanisme se nomme "redirection". Il a également la possibilité de fermer ses entrées/sorties standard, temporairement ou définitivement, s'il ne veut plus les utiliser. Ça sera par exemple le cas d'un programme qui ne fait aucune saisie et qui inscrit ses résultats dans un fichier, par exemple un programme qui calcule des nombres premiers. Les redirections peuvent être faites soit dans le programme lui-même, soit par l'interprète de commandes au moment du lancement de la commande.

4.4.6 Les tubes

Il est assez courant que l'utilisateur d'Unix ait envie que l'entrée standard d'un processus soit la sortie standard d'un autre. Par exemple, l'utilisateur souhaite connaître le nombre de fichiers et répertoires contenus dans un répertoire donné. Sous Unix, il dispose d'une commande qui permet d'afficher le contenu d'un répertoire, à raison d'une entrée par ligne. Il dispose d'une autre commande qui permet de compter le nombre de ligne contenues sur l'entrée standard. Une solution à son problème est de lancer la première commande en redirigeant sa sortie standard vers un fichier, puis la seconde commande avec comme entrée standard ce fichier. Après exécution de la seconde commande, on efface le fichier. Cette façon de faire présente de nombreux désavantages très importants :

- Il y a création de fichier, et donc recours à des mécanismes d'entrées/sorties complexes sur un périphérique qui est probablement un disque, donc très lent,
- il faut penser à effacer le fichier
- le fichier peut être de taille importante

Pourquoi mettre des informations dans un fichier, alors qu'elles vont être supprimées ? Les deux notions importantes dans ce type de problème sont la notion de producteur (le premier processus produit des informations), et la notion de consommateur (le second consomme les informations produites par le premier). Sous Unix on dispose d'un mécanisme qui permet de connecter les deux commandes en disant que l'entrée standard de la seconde est la sortie standard de la première. Ce mécanisme permet de réaliser ce qu'on souhaite. Cette connection se fait par l'intermédiaire d'un "tube", sur le modèle producteur/consommateur : les informations produites par le premier processus sont utilisées au fur et à mesure de leur production par le consommateur. Le producteur et le consommateur sont exécutés "parallèlement", l'un à l'entrée du tube et l'autre à la sortie. Les informations sont produites et consommées de manière synchrones (avec un petit décalage permis) : l'idée est que les informations sont consommées au fur et à mesure de leur production. Si le consommateur est trop lent pour le producteur, ce dernier l'attend. Si c'est l'inverse, c'est le consommateur qui attend.

4.4.7 Les processus en tâche de fond

Un processus en "tâche de fond" est un processus qui, comme son nom l'indique, fait son calcul en arrière plan, c'est-à-dire sans utiliser le clavier comme entrée standard (par contre, à moins de mentionner le contraire, la sortie standard reste l'écran). L'entrée standard d'un

processus qui s'exécute en arrière plan est soit redirigée vers un autre périphérique, soit fermée. Cette redirection est faite soit automatiquement par l'interprète de commande, en lui spécifiant qu'on veut lancer le programme en tâche de fond, soit par le programme lui-même.

4.4.8 Le répertoire de travail

À chaque processus est attaché un répertoire "de travail", qui peut varier au cours de la vie du processus. Cependant, la majorité des processus ne changent pas leur répertoire de travail. Pour bien comprendre de quoi il s'agit, prenons un exemple simple, celui d'un processus interactif en mode texte qui permet à l'utilisateur de se promener dans le SGF (c'est le cas, par exemple, des interprètes de commande) et d'y faire des opérations (les interprètes de commande font partie des rares processus qui changent leur répertoire de travail). L'utilisateur choisit un répertoire, dans lequel il se place. Le processus lui permet alors de manipuler les entrées de ce répertoire sans avoir à en donner les références absolues : les références qu'il donne sont relatives au répertoire dans lequel il se trouve à l'instant des manipulations. Ce répertoire s'appelle le "répertoire de travail" du processus, ou le "répertoire courant". Quand le processus souhaite accéder à une entrée du système de fichiers de manière relative (c'est-à-dire par une référence ne commençant pas par un /), elle le fait relativement à son répertoire de travail. Supposons par exemple que le répertoire courant du processus soit `/usr/lib`, et que l'utilisateur souhaite accéder, à partir de là, au fichier de référence absolue `/usr/bin/zip`. Il pourra employer la référence relative `../bin/zip`.

4.4.9 Les variables d'environnement

Elles permettent aux processus de se transmettre des valeurs par filiation. Ces valeurs sont des chaînes de caractères. Il n'existe pas de variable d'environnement qui soit de type numérique. Quand on souhaite qu'une variable d'environnement ait une valeur numérique, on doit donc convertir cette valeur en chaîne de caractères. Les variables d'environnement ont des noms, qui sont aussi des chaînes de caractères, avec quelques conventions (pas de caractères spéciaux, pas d'espace, ...) d'usage pour les noms de variables en informatique. Un processus peut à tout moment consulter, modifier, supprimer ou ajouter une variable de son d'environnement. Comme presque toutes les caractéristiques d'un processus, les variables d'environnement seront transmises aux processus qu'il crée au moment de la duplication. Comme exemple de variable d'environnement, citons `PATH`, qui contient la liste des répertoires ordonnée dans laquelle on doit aller chercher les commandes à exécuter quand la localisation de leur exécutable n'est pas explicitement donnée. Voilà une valeur typique pour `PATH` : `/usr/java/jdk1.3/bin: /bin: /usr/local/bin: /usr/bin/X11: /usr/contrib/bin/X11: /usr/bin: /home/divers/jdk1.2.2/bin: ..` Une autre variable typique est `HOME`. Elle contient le répertoire de login de l'utilisateur propriétaire du processus. Un exemple de valeur pour `HOME` est `/home/bedon`. Signalons pour terminer une autre variable d'environnement importante : `PWD`, dont la valeur est le répertoire de

travail du processus. Finissons par la variable `TERM`, dont la valeur est consultée par les programmes qui veulent connaître les caractéristiques du terminal à partir duquel elles sont lancées, par exemple pour en connaître le nombre de colonnes et de lignes, ou bien encore pour s’informer des capacités du terminal à déplacer son curseur ou à faire de la couleur. Les terminaux `vt100` et `vt52` sont des terminaux textes très répandus. De nombreuses applications (de connexion à distance, par exemple, les émulent). Le terminal `xterm` est le terminal sous X le plus répandu sous X. Les valeurs de `TERM` sont alors `vt100`, `vt52`, ou `xterm` selon le cas.

4.5 L’interface homme-machine

Il existe plusieurs types d’IHM sous Unix, graphiques ou textuelles.

4.5.1 Interface graphique

La base de toute interface graphique sous Unix se nomme X. C’est un système qui offre des possibilités de base pour faire du graphisme, ouvrir des fenêtres, . . . Il a été développé au MIT. Son nom vient du fait qu’il est le successeur d’un autre système graphique, toujours développé au MIT, qui s’appelait W pour “Window”. X est utilisé par d’autres programmes, plus simples, qui permettent eux de définir un environnement de travail à base de fenêtres graphiques. Ces programmes sont appelés “gestionnaires de fenêtres”. Il en existe foison (`fwm`, `fvwm`, `kde`, . . .), qui diffèrent par leur comportement et l’aspect. Leurs comportement tend toutefois vers une normalisation à la Windows.

4.5.2 Interprète de commandes

Ils sont indispensables pour lancer des commandes à partir du clavier et pour le traitement par lots. Comme les interfaces graphiques, ils sont foison. Les plus connus sont :

- le shell `sh`, l’ancêtre, la base de tout
- le C-shell `csh` (du shell amélioré avec une syntaxe à la C)
- le TC-shell `tcsh` (toujours plus fort)
- le korn-shell `ksh`
- le Bourne Again shell `bash`, shell de base amélioré

Ils diffèrent par les fonctionnalités qu’ils offrent, et aussi parfois malheureusement par leur syntaxe. Une description assez complète du `bash` est l’objet de la section suivante.

4.5.3 Les applications

Le nombre d’applications qu’Unix supporte est proportionnel au grand âge de ce système. Signalons simplement l’existence de jeux, d’applications bureautiques, d’applications scientifiques, d’applications multimédia, . . . Nous ne décrivons ici que deux qui vous seront très utiles : Emacs, un éditeur de textes, et Netscape, un navigateur Web et gestionnaire de courrier électronique. Signalons aussi au passage le processeur de textes \LaTeX , très utilisé dans le monde scientifique, et qui a été utilisé pour écrire ce document.

L'éditeur de textes

L'éditeur de textes sert à visualiser le contenu d'un ou plusieurs fichiers texte, à le modifier, à le sauvegarder, à en créer un nouveau. C'est un outil indispensable au programmeur.

Emacs est parmi les éditeurs de textes un de ceux qui réalisent le plus d'opérations. C'est un outil très puissant, qui met à la disposition de son utilisateur beaucoup de fonctions permettant de lui faciliter sa tâche. Emacs est également un gros consommateur de ressources. Ceci veut dire que le lancement simultané de plusieurs Emacs peut entraîner un ralentissement notable des performances de la machine. L'alternative proposée par Emacs est la possibilité d'ouvrir plusieurs fichiers à la fois, dans une ou plusieurs fenêtres, mais toutes gérées par le même processus.

Citons comme exemples de fonctionnalités :

- l'aide à la présentation de sources de programmes (indentation, coloriage, ...)
- une aide à la compilation et à la correction des erreurs éventuellement détectées
- la correction orthographique
- la possibilité d'ajouter des modules fournissant de nouvelles fonctionnalités à l'éditeur
- un langage de programmation (Emacs-lisp) permettant d'écrire soi-même de nouvelles fonctionnalités pour l'éditeur
- une documentation hypertexte sur la plupart des outils GNU
- ...

Emacs peut se piloter à partir de la souris, ce qui en rend l'utilisation particulièrement conviviale. Pour les utilisateurs plus expérimentés, il existe des raccourcis clavier qui diminuent considérablement le temps passé à réaliser les opérations habituelles, tels le chargement ou la sauvegarde d'un fichier. Un aide mémoire trouvé sur www.refcards.com (ce site en contient pour plusieurs autres logiciels) est donné en annexe.

Le Logiciel de navigation Netscape

Le logiciel de navigation Netscape permet d'utiliser la toile (le Web) pour y chercher des informations dans le monde entier.

Les pages auxquelles on accède en surfant sont toutes écrites dans un langage de description de documents hypertextes, appelé HTML — *HyperText Markup Language*. La syntaxe de ce langage est si simple qu'il n'est pas nécessaire d'être informaticien pour créer des pages directement en HTML. On peut aussi créer des pages Web sans manipuler du tout ce langage, puisqu'il existe des logiciels spécialisés dans la création de sites Web où tout se fait à la souris. On peut, à tout moment, consulter les sources de la page Web courante du navigateur en consultant dans le menu *view* l'article *document source*.

Signalons enfin que le Web contient tellement d'informations de source non contrôlée qu'on s'y perd facilement... Il faut donc toujours naviguer de façon dirigée (jamais au hasard).

5 Les principales commandes d'Unix

Sous les interprètes de commandes d'Unix, la syntaxe d'une commande est de la forme :

- nom de la commande
- arguments
- options
- redirections
- indicateur de tâche de fond éventuel

Les options sont différenciées des arguments en les faisant préfixées par un tiret (-). L'indicateur de tâche de fond est toujours &. La syntaxe pour les redirections de sortie erreur standard varie avec les interprètes de commandes, mais la redirection de l'entrée standard est toujours notée par < et celle de la sortie standard par >. L'utilisation d'un tube est toujours notée par le caractère |.

Par exemple, la commande

```
ls /tmp /usr -i -a
```

indique qu'on veut lancer la commande `ls`, qui affiche le contenu des répertoires fournis en argument sur sa sortie standard, sur les répertoires `/tmp` et `/usr` avec pour options `-a`, qui signifie qu'on veut que toutes les entrées du répertoire soient affichées, et `-i`, qui signifie qu'on souhaite que le numéro d'inode associé à l'entrée du répertoire soit également affiché.

Pour rediriger la sortie standard de cette commande vers le fichier `/tmp/toto`, la syntaxe est

```
ls /tmp /usr -i -a >/tmp/toto
```

Le fichier `/tmp/toto` sera créé s'il n'existe pas déjà. S'il existe déjà, son contenu sera remplacé par la sortie de la commande `ls /tmp /usr -i -a`.

Nous donnons maintenant un exemple d'utilisation des tubes. La commande `wc` (Word Count) permet de compter, suivant les options qu'on lui donne, les caractères, les lignes et les mots de son entrée standard. Elle envoie le résultat du comptage sur sa sortie standard. La commande `ls -a /tmp` affiche les entrées du répertoire `/tmp`. Quand la sortie de la commande est redirigée, elle contient une entrée par ligne. Pour compter le nombre d'entrée du répertoire, il suffit donc de compter le nombre de lignes par la commande `ls -a /tmp | wc -l`.

5.1 Commandes pour manipuler le système de fichiers

Quand ça a un sens, la plupart de ces commandes peuvent effectuer des actions récursivement, avec l'option `-r` ou `-R`. Bien entendu, pour effectuer une opération sur une entrée du système de fichiers, il faut en avoir le droit !

- `ls` permet de consulter le contenu (les entrées) d'un ou de plusieurs répertoires dont les noms sont fournis en argument. Par défaut d'argument, elle travaille sur

le répertoire courant. Cette commande dispose de nombreuses options, qui permettent d'avoir plus ou moins d'informations que chaque entrée, ou bien encore de faire un parcours récursif du système de fichiers.

- **mv** permet de déplacer ou de changer de nom une (ou plusieurs) entrée d'un répertoire. Si le dernier argument est un répertoire, toutes les entrées spécifiées avant seront déplacées dans le répertoire. Sinon, s'il n'y a que deux arguments sur la ligne de commande, le nom du premier devient le second.
- **cp** a une syntaxe similaire à **mv**, mais elle réalise une copie plutôt qu'un déplacement. Attention, copier un répertoire est une opération récursive, contrairement à changer le nom d'un répertoire.
- **rm** efface les fichiers spécifiés en argument. Attention, c'est une opération irréversible, et beaucoup d'utilisateurs en ont été victime.
- **rmdir** efface les répertoires vides spécifiés en argument.
- **mkdir** permet de créer les répertoires dont les noms sont données en argument.
- **cd** permet de se promener dans l'arborescence du système de fichier : la commande permet de changer de répertoire de travail.
- **chmod** permet de changer les droits sur une entrée du système de fichiers. Bien entendu, il faut être propriétaire de cette entrée pour pouvoir en changer les droits.

5.2 Commandes pour manipuler les processus

- **ps** permet d'obtenir la liste des processus existant sur le système au moment du lancement de la commande
- **kill** permet d'envoyer un signal à un processus. En particulier, quand un processus devient incontrôlable, on lui envoie le signal **SIGKILL** pour qu'il se termine immédiatement. Pour envoyer un signal à un processus, il faut en être propriétaire.
- **nice** permet de modifier la priorité d'un processus. Bien entendu, cette modification obéit à des contraintes de sécurité de bon sens : il faut être propriétaire du processus, et il existe un plafond de priorité.
- **nohup** permet de lancer des processus de telle manière qu'ils ignorent la déconnection de leur terminal de rattachement.
- **top** permet de voir les processus les plus consommateurs de CPU, de se rendre compte de leur ordonnancement, état, ...

5.3 Commandes diverses

- **echo** copie ses arguments sur la sortie standard
- **cat** copie son entrée standard sur sa sortie standard quand elle n'a pas d'argument. Quand elle en a, ce sont des fichiers qui prennent la place de l'entrée standard.
- **wc** permet de compter les caractères, mots et lignes de son entrée standard
- **lp** (ou **lpr**, suivant la version d'Unix) permet de faire des impressions
- **more** permet, quand on lance une commande dont le résultat comprend trop de lignes pour tenir sur l'écran du terminal, de contrôler le défilement du résultat (en

- particulier de l'arrêter)
- **less** est une version plus sophistiqué de **more** (contrairement à ce qu'indiquent les noms, qui font croire le contraire), qui permet en particulier le retour en arrière.
- **grep** est une commande qui permet de sélectionner des lignes de l'entrée standard (ou des fichiers donnés en argument) en fonction des caractères qu'elles contiennent.
- **which** est une commande prenant en argument le nom d'une commande et permettant de localiser l'exécutable de cette commande. Elle est très utilisée quand plusieurs commandes différentes ont le même nom, et qu'on ne sait plus laquelle on exécute.
- **eval** évalue son argument
- **passwd** permet de changer son mot de passe

Finalement, la commande la plus utile de toutes : **man** permet d'obtenir le manuel sur une commande, un appel système, un format de fichier, ... dont le nom est fourni en argument. Pour obtenir de l'aide sur la commande **man** : **man man** !! Pour plus de détails sur une des commandes précédemment citées, fournir son nom à **man** !!

6 Le bash

Cette section est directement tirée de [Pro97].

Le **bash** est un interprète de commandes puissant et très utilisé. Dans cette partie, nous nous inspirons grandement du manuel (**man**) fourni avec le *shell* **bash**.

6.1 bash : Introduction et rappels

bash (*Bourne Again Shell*) est un *shell* —un interprète de commandes— compatible avec **sh** (*Bourne Shell*, le premier *shell* d'Unix) qui exécute des commandes lues sur l'entrée standard ou dans un fichier. **bash** inclut notamment des caractéristiques des *shells* **ksh** (*Korn Shell*) et **csh** (*C-Shell*). Il se veut une implémentation conforme des spécifications Posix IEEE en matière de *shells* et leurs outils.

Quelques options

- Quelques options intéressantes de la commande **bash** :
- **-norc** ne lit pas le fichier **.bashrc** ;
 - **-noprofile** ne lit pas les fichiers d'initialisation des *login shells* ;
 - **-rcfile <fic>** lit le fichier **fic** à la place de **.bashrc** ;
 - **-version** affiche la version de **bash** avant son exécution ;
 - **-login** se comporte comme un *login shell*.

Définitions préliminaires

- un **blanc** est un espace ou une tabulation ;
- un **mot** est une séquence de caractères d'un seul tenant ;

- un **séparateur** est un caractère non-déspecialisé qui sépare les mots entre eux :
| & ; () < > espace tabulation ;
- un **opérateur de contrôle** est un objet qui réalise une fonction particulière :
|| & && ; ;; () | return ;
- un **mot réservé** est un mot qui possède un sens bien défini pour **bash** : ! case do done elif else esac fi for function if in select then until while { }, ou encore le premier mot d'une commande simple, ainsi que le troisième mot d'une commande for ou case.

Grammaire de bash

bash, ainsi que les autres *shells*, ne reconnaît les commandes qu'on lui soumet que si leur syntaxe est correcte : c'est ce qu'on appelle sa *grammaire*, que nous décrivons à présent.

Une **commande simple** est une séquence de mots séparés par des blancs et des redirections et terminée par un opérateur de contrôle. Le premier mot indique la commande à exécuter, alors que les autres sont passés en arguments de la commande. La valeur de retour d'une commande simple est son code de retour, ou $128 + n$ si la commande a été interrompue par le signal n .

Un **pipeline** est une séquence de commandes séparées par des | qui répond à la syntaxe :

```
<commande1> [ | <commande2> | .. | <commanden> ]
```

Le *shell* attend la terminaison de toutes les commandes avant de récupérer le code de retour de la dernière. Notez qu'une commande simple est aussi appelée *pipeline* ($n=1$).

Une **liste** est une séquence d'un ou plusieurs *pipelines* séparés par un des opérateurs ; , & && ou || et terminée par ; , & ou return. Notez que && et || sont prioritaires sur ; et & . Leurs significations sont respectivement le ET et le OU sur les listes.

Ainsi une liste ET est de la forme <commande1> && <commande2> et implique que <commande2> n'est exécutée que si <commande1> a pour code de retour 0. Il en va de même pour une liste OU, où <commande2> n'est exécutée que si <commande1> produit un code de retour non nul. La valeur de retour de ces listes est le code de la dernière commande.

Une **commande composée** est au choix :

- (<liste>) : la <liste> est exécutée dans un sous-*shell*. Le code de retour est celui de <liste> ;
- { <list>; } : la <liste> est exécutée dans le *shell* courant (regroupement de commandes) ;
- une instruction **for** (voir section sur les structures de contrôle) ;
- une instruction **select** (voir section sur les structures de contrôle) ;
- une instruction **case** (voir section sur les structures de contrôle) ;
- une instruction **if** (voir section sur les structures de contrôle) ;
- une instruction **while** (voir section sur les structures de contrôle) ;
- une instruction **until** (voir section sur les structures de contrôle) ;

– une fonction (voir section sur les fonctions).

Un **commentaire** commence par le caractère **#** : tout ce qui suit dans la ligne courante est ignoré par **bash**.

Commandes **bash** internes

Certaines commandes de **bash** ne sont pas des processus : elles appartiennent au langage **bash** lui-même. On les appelle *commandes internes* — *builtin commands*. La liste de ces commandes est la suivante :

<code>:</code> [<code><arguments></code>]	Sans effet, si ce n'est l'expansion des arguments et les redirections.
<code>.</code> <code><fichier></code> [<code><arguments></code>] <code>source</code> <code><fichier></code> [<code><arguments></code>]	Lit et exécute les commandes du fichier dans l'environnement courant et retourne le code de la dernière commande, 0 s'il n'y en a pas. Les arguments deviennent les paramètres de position pour le fichier.
<code>alias</code> [<code><nom></code> [<code>=<valeur></code>]]	Sans argument, donne la liste des alias existants. Avec un nom, donne la valeur de l'alias. Avec une affectation, positionne l'alias. Le retour d' <code>alias</code> est toujours vrai, sauf si on demande la valeur d'un alias qui n'existe pas.
<code>bg</code> [<code><job></code>]	Voir le <i>job control</i> .
<code>bind</code>	Documentez-vous.
<code>break</code> [<code><nombre></code>]	Sort d'une boucle <code>for</code> , <code>while</code> ou <code>until</code> . Si un nombre est spécifié, traverse <code><nombre></code> boucles. La valeur de retour est 0, sauf si <code>break</code> n'est pas appelé dans une boucle.
<code>builtin</code>	Documentez-vous.
<code>cd</code> [<code><repertoire></code>]	Déjà vu.
<code>command</code> [<code>-pVv</code>] <code><commande></code> [<code><arguments></code>]	Documentez-vous.
<code>continue</code> [<code><nombre></code>]	Reprend l'itération suivante d'une boucle <code>for</code> , <code>while</code> , <code>until</code> . Si <code><nombre></code> est spécifié, reprend à la <code><nombre></code> ième itération. Si <code><nombre></code> est supérieur au nombre total de boucles, reprend uniquement à la dernière boucle. Le code de retour est 0, sauf si <code>continue</code> n'est pas exécutée dans une boucle.

`declare [-frxi] [<nom>[=<valeur>]]`
`type [-frxi] [<nom>[=<valeur>]]`

`dirs`

`echo [-neE] [<arguments>]`

`enable [-n] [-all] [<nom> ..]`

`eval [<arguments>]`

`exec [[-] <commande> [<arguments>]]`

`exit [<nombre>]`

`export [-nf] [<nom>[=<mot>]]`
`export -p`

`fc`

`fg [<job>]`

`getopts`

Déclare des variables ou leur donne des attributs. Si un nom n'est pas spécifié, affiche les attributs des variables. `-f` pour les fonctions ; `-r` pour positionner en lecture seule ; `-x` pour marquer la variable en vue d'une exportation ; `-i` pour en faire une variable arithmétique (sujette à l'expansion arithmétique).

Permet de mémoriser les répertoires traversés (voir `pushd`). Documentez-vous. Affiche les arguments sur la sortie standard, séparés par des espaces. L'option `-n` enlève les retours chariots.

Active ou désactive les commandes internes du *shell*. Avec l'option `-n`, chaque `<nom>` est désactivé, sinon tous les `<noms>` sont activés. Sans argument, affiche la liste des commandes actives. La valeur de retour est 0 sauf si `<nom>` n'est pas valide.

Les arguments sont lus et concaténés en une seule commande. Cette commande est ensuite lue et exécutée par le *shell*. Le code de retour est celui de la commande exécutée.

Si une commande est spécifiée, elle remplace le *shell* courant. Aucun processus n'est créé. Les arguments sont ceux de la commande. Si le premier argument est un `-`, le *shell* le place dans le zéro-ième argument de `<commande>`. Si la commande n'est pas spécifiée, toutes les redirections prennent effet dans le *shell* courant, et le code de retour est 0.

Sort du *shell*, avec éventuellement un code de retour `<nombre>`.

Les `<noms>` spécifiés sont marqués pour l'exportation. L'option `-f` spécifie des fonctions, `-n` retire les `<noms>` pour l'exportation. `export -p` affiche les variables qui seront exportées.

Documentez-vous.

Voir le *job control*.

Concerne les paramètres de position. Documentez-vous.

<pre>hash [-r] [<nom>]</pre>	<p>Se souvient pour chaque <nom>, du chemin de la commande (évite les recherches laborieuses). L'option <code>-r</code> détruit les mémorisations.</p>
<pre>help [<motif>]</pre>	<p>Affiche des informations sur les commandes internes (essayez de vous en souvenir). Le <motif> permet de donner le début d'une commande.</p>
<pre>history -rwan [<fichier>] history [<nombre>]</pre>	<p>Sans option, affiche l'historique des commandes avec leurs numéros. L'argument <nombre> spécifie le nombre de lignes à afficher. Documentez-vous pour les options.</p>
<pre>jobs [-lnp] [<jobs>] jobs -x <commande> [<arguments>] [<jobs>]</pre>	<p>Le premier donne la liste des <i>jobs</i> actifs. L'option <code>-l</code> liste en plus les PID. L'option <code>-p</code> liste uniquement le PID du processus en tête du groupe. L'option <code>-n</code> liste uniquement les <i>jobs</i> qui ont changé de statut depuis la dernière notification. L'option <code>-x</code> remplace tous les <i>jobs</i> trouvés dans <commande> ou <arguments> par l'identification du groupe processus et exécute <commande> en lui passant les <arguments>.</p>
<pre>kill [-s <sig> -<sig>] [<pid> <job>] kill [-l] [<signal>]</pre>	<p>Envoie le signal <sig> aux processus référencés soit par leur <pid>, soit par le <job>. Le signal par défaut est <code>SIGTERM</code>. L'argument <code>-l</code> donne la liste des signaux disponibles.</p>
<pre>let <argument> [<arguments>]</pre>	<p>Chaque <argument> est une expression arithmétique à évaluer. Si le dernier <argument> vaut 0, retourne 1, sinon retourne 0.</p>
<pre>local [<nom>[=<valeur>]...]</pre>	<p>Pour chaque argument, crée une variable locale appelée <nom> et lui assigne une <valeur>. A l'intérieur d'une fonction, la portée de la variable locale se restreint au corps de la fonction (retour =0). Partout ailleurs, une erreur est produite (retour ≠ 0). Sans opérande, affiche la liste des variables locales.</p>
<pre>logout</pre>	<p>Sort d'un <i>login shell</i>.</p>

popd [+/-<nombre>]

Enlève les entrées de la pile des répertoires. Sans argument, enlève le sommet de pile et exécute un `cd` sur le nouveau sommet. `+<nombre>` enlève la `<nombre>`ième entrée à partir de la gauche quand on exécute `dirs` ; `-<nombre>` la même chose à partir de la droite. Code de retour 0 si commande correcte.

pushd [<repertoire>]

pushd +/-<nombre>

Ajoute un répertoire à la pile des répertoires, ou effectue une rotation de la pile en positionnant le sommet au répertoire courant. `+<nombre>` et `-<nombre>` effectuent une rotation du `<nombre>`ième répertoire (voir `popd`). `<repertoire>` ajoute `<repertoire>` en sommet de pile, le transformant en nouveau répertoire de travail. Code de retour 0 si commande correcte.

pwd

Affiche la référence absolue du répertoire de travail. Avec l'option `-P`, le chemin affiché ne contient pas les liens symboliques.

read [-r] [<nom> ...]

Une ligne est lue sur l'entrée standard, et le premier mot reçoit la variable `<nom>`, etc. Si `<nom>` n'apparaît pas, la ligne lue est assignée à la variable `REPLY`. Le code de retour est 0, tant qu'une fin de fichier n'est pas rencontrée. L'option `-r` empêche d'ignorer les `\<newline>`.

readonly [-f] [<nom> ...]

readonly -p

Les `<noms>` donnés sont marqués en lecture seule. Si l'option `-f` est spécifiée, la liste des variables positionnées en lecture seule apparaît. Une erreur est produite si une option ou un nom n'est pas valide.

return [<nombre>]

Provoque la terminaison d'une fonction avec la valeur de retour spécifiée. Sans valeur spécifiée, le retour est positionné à celui de la dernière commande de la fonction. Peut aussi être utilisé en dehors d'une fonction, ce qui provoque l'arrêt du script avec le code de retour spécifié.

set [--abefhkmnptuvxldCHP]
[-o <option>] [<arguments>]

- a marque automatiquement les variables qui sont modifiées ou créées pour l'exportation.
- b demande le rapport immédiat des processus en arrière-plan terminés.
- e sort immédiatement si une commande ne produit pas un code de retour égal à 0.
- f enlève l'expansion des noms de chemins.
- h localise et se souvient des commandes fonctions dès leur définition.
- k Tous les arguments mots-clefs sont placés dans l'environnement pour une commande, pas seulement ceux qui précèdent le nom de la commande.
- m Mode *moniteur*. Le *job control* est activé.

- n lit les commandes mais sans les exécuter.
- o [<option>] Documentez-vous. L'<option> correspond souvent à une des autres options de cette liste. Sans argument, affiche les options positionnées.
- p Active le mode *privilegié*...
- t Sort après la lecture et l'exécution de la commande.
- u Traite les variables non positionnées comme une erreur au cours de l'expansion. Provoque la sortie du *shell* en mode script.
- v affiche les lignes du *shell* telles qu'elles sont lues.
- x Après expansion de chaque commande simple, affiche la valeur de PS4, suivie de la commande et de ses arguments.
- l Sauve et restaure la liaison de <nom> dans une boucle `for <nom> ..`
- d Désactive la fonction de hachage.
- C Impose l'écrasement explicite des fichiers.
- H Positionne le style ! pour l'historique.

- P Ne suit pas les liens symboliques lors de l'exécution de commandes telle `cd`.
- S'il n'y a pas de drapeau, les paramètres de position sont enlevés. Sinon, ils sont positionnés aux <arguments>, même si l'un d'eux commence par un -.
- Signale la fin des options : tous les arguments restants deviennent des paramètres de position. Les options -x et -v sont désactivées.

`shift [<nombre>]`

Les paramètres de position à partir de <nombre> +1 ... sont renommés en \$1 Par défaut, <nombre> vaut 1. Retour correct si <nombre> est valide.

suspend [-f]

test [<expr>]
[<expr>]

Suspend l'exécution de ce *shell*. L'option -f demande de ne pas afficher d'erreur si c'est un *login shell*.

La deuxième paire de crochets est valide. Retourne 0 ou 1 selon l'évaluation de l'expression conditionnelle <expr>. Les expressions peuvent être unaires ou binaires, les expressions unaires étant souvent destinées à l'examen du statut d'un fichier. Si le fichier est de la forme /dev/fd/<nombre>, le descripteur <nombre> est vérifié.

-d <fichier> vrai si le fichier est un répertoire.

-e <fichier> vrai si le fichier existe.

-f <fichier> vrai si le fichier est un fichier régulier.

-g <fichier> vrai si le fichier existe et a le *setgid* bit.

-k <fichier> vrai si le fichier a le *sticky* bit positionné.

-L <fichier> vrai si le fichier existe et est un lien symbolique.

-p <fichier> vrai si le fichier existe et est un tube nommé.

-r <fichier> vrai si le fichier est lisible.

-s <fichier> vrai si le fichier a une taille positive.

-S <fichier> vrai si le fichier existe et est une *socket*.

-t <descripteur> vrai si le descripteur est ouvert sur un terminal.

-u <fichier> vrai si le fichier a le *setuid* bit.

-w <fichier> vrai si le fichier est écrasable.

-x <fichier> vrai si le fichier est exécutable.
-0 <fichier> vrai si le fichier est possédé par l'utilisateur effectif.
-G <fichier> vrai si le fichier est possédé par le groupe effectif.
<fic1> -nt <fic2> vrai si <fic1> est plus récent que <fic2>.
<fic1> -ot <fic2> vrai si <fic1> est plus vieux que <fic2>.
<fic1> -ef <fic2> vrai si les deux fichiers ont même disque logique et même i-nœud.
-z <chaîne> vrai si la longueur de la chaîne est zéro.
-n <chaîne> / <chaîne> vrai si la longueur de la chaîne n'est pas zéro.

<ch1> = <ch2> vrai si les deux chaînes sont égales.
<ch1> != <ch2> vrai si les deux chaînes sont différentes.
!<expr> vrai si l'expression est fausse.
<expr1> -a <expr2> vrai si les deux expressions sont vraies (et).
<expr1> -o <expr2> vrai si l'une des deux expressions est vraie (ou).
<expr1> <op> <expr2> vrai si l'opérateur est vérifié : -eq, -ne, -lt, -le, -gt, -ge de significations respectives =, ≠, <, ≤, >, ≥.

times

Affiche les temps cumulés utilisateur et système pour le *shell* et pour les processus du *shell*.

trap

Captage de signaux. Documentez-vous.

`type [-all] [-type|-path] <nom> [<nom> ..]` Sans option, indique comment chaque `<nom>` serait interprété s'il était utilisé comme commande. Si l'option `-type` est positionnée, affiche une phrase qui est soit un alias, soit un mot-clef, soit une fonction, soit une commande interne, soit un fichier en fonction de la valeur de `<nom>`. Si `<nom>` n'est pas trouvé, rien ne s'affiche. Si l'option `-path` est positionnée, affiche ou retourne le nom du fichier disque qui serait exécuté si `<nom>` est une commande valide, sinon rien. L'option `-all` affiche tous les endroits qui contiennent un nom exécutable `<nom>` (la table de hachage n'est pas consultée).

`ulimit [-SHacdfmstpnuv [<limite>]]`

Fournit un contrôle sur les ressources disponibles pour le *shell* et les processus qu'il lance. La valeur de `<limite>` peut être un nombre dans l'unité spécifiée par la ressource ou encore la valeur `<unlimited>`. Les options `S` et `H` spécifient que la limite matérielle ou logicielle est positionnée pour la ressource (par défaut : `H`). Si `<limite>` est omis, la valeur courante de la limite logicielle / matérielle est affichée. Lorsque plus d'une ressource est spécifiée, les unités sont affichées. Les autres options sont : `-a` toutes les limites sont affichées. `-c` taille maximum des fichiers *core*. `-d` taille maximum du segment de données d'un processus. `-f` taille maximum des fichiers créés par le *shell*. `-m` taille maximum du *set* résident. `-s` taille maximum de pile. `-t` nombre maximum de secondes de CPU. `-p` taille d'un tube en blocs de 512 octets. `-n` nombre maximum de descripteurs de fichiers ouverts. `-u` nombre maximum de processus par utilisateur. `-v` mémoire virtuelle maximum disponible pour le *shell*. Si `<limite>` est fournie, c'est la nouvelle valeur de la ressource spécifiée. Sans option, `-f` est positionnée par défaut. Le retour est différent de 0 en cas de paramètres non valides.

<code>umask [-S] [<mode>]</code>	La création de fichiers se fait avec le masque (<mode>) spécifié. Si le mode est omis, la valeur par défaut du masque est affichée soit explicitement (option <code>-S</code>), soit en octal.
<code>unalias [-a] [<nom> ...]</code>	Enlève chaque <nom> de la liste d'alias. L'option <code>-a</code> détruit tous les alias.
<code>unset [-fv] [<nom> ...]</code>	Pour chaque nom, détruit la variable correspondante, ou si l'option <code>-f</code> est positionnée, la fonction. Notez que certaines variables de l'environnement ne peuvent être effacées. Le code de retour est vrai sauf si le <nom> n'est pas valide ou ne peut être effacé.
<code>wait [<pid>]</code>	Attend la terminaison du processus spécifié et retourne son statut. <pid> peut au choix être un PID ou un numéro de <i>job</i> . Si c'est un numéro de <i>job</i> , attend la terminaison de tous les processus du même <i>pipeline</i> . Si <pid> n'est pas fourni, affiche tous les processus fils qui sont attendus et retourne 0.

Environnement de `bash` et variables

Lorsque vous lancez un `bash`, vous disposez d'un certain nombre de variables accessibles en les faisant précéder du caractère `$`. Exemple :

```
fillmore> echo $USER
prosper
fillmore>
```

Nous donnons ci-dessous les variables courantes :

<code>PPID</code>	numéro de processus du <i>shell</i> père
<code>PWD</code>	répertoire de travail courant
<code>UID</code>	identification de l'utilisateur actuel
<code>BASH</code>	chemin d'accès à la commande <code>bash</code>
<code>BASH_VERSION</code>	version du <code>bash</code>
<code>HOSTTYPE</code>	type de la machine utilisée
<code>PATH</code>	ensemble des chemins de recherche des commandes invoquées : chemins dans l'arborescence du disque, séparés par des :
<code>HOME</code>	répertoire principal de l'utilisateur
<code>CDPATH</code>	ensemble des répertoires parcourus pour la commande <code>cd</code> : même syntaxe que <code>PATH</code>

ENV	fichier contenant des commandes d'initialisation pour les scripts. Attention : PATH n'est pas utilisé dans ce fichier <i>shells</i>
PS1	<i>prompt</i> principal
PS2	<i>prompt</i> secondaire : signifie l'attente de données pour compléter une commande
PS3	<i>prompt</i> pour l'instruction select
PS4	<i>prompt</i> affiché avant chaque commande durant le traçage d'une exécution. La réplication de ce <i>prompt</i> indique le niveau d'imbrication

D'autres variables, d'intérêt secondaire :

OLDPWD	ancien répertoire de travail (<code>cd -</code>)
REPLY	ligne lue par une commande interne du <i>shell</i>
EUID	identification de l'utilisateur effectif
SHLVL	niveau de sous- <i>shell</i>
RANDOM	un entier au hasard
SECONDS	nombre de secondes depuis le démarrage du <i>shell</i>
LINENO	numéro de ligne dans un script
HISTCMD	numéro de la commande dans l'historique
OSTYPE	type du système d'exploitation utilisé
IFS	ensemble des séparateurs de mots après expansion
MAIL	fichier qui contient le courrier
MAILCHECK	donne les intervalles (en secondes) de temps pour vérifier l'arrivée de courriers
HISTSIZE	nombre de commandes mémorisables dans l'historique
HISTFILE	fichier contenant l'historique des commandes
TMOUT	temps d'attente d'une commande avant la terminaison automatique de bash
FIGIGNORE	suite de suffixes à ignorer pour la complétion de noms de fichiers. Les suffixes sont séparés par des :
INPUTRC	fichier d'initialisation du bash
notify	si elle est positionnée, bash affiche immédiatement les <i>jobs</i> en arrière-plan terminés
nolinks	si elle est positionnée, bash ne suit pas les liens symboliques qui envoient sur d'autres répertoires que le répertoire courant
noclobber	si elle est positionnée, bash empêche l'écrasement de fichiers par les redirections qui ne sont pas suivies du caractère
no_exit_on_failed_exec	si elle est positionnée, bash ne termine pas si un recouvrement de lui-même a échoué

Notons aussi qu'en plus de certaines facilités offertes par **bash** comme les mécanismes d'historique, d'alias, ou encore la pile des répertoires, il existe aussi une fonctionnalité très pratique appelée la *complétion* : le *shell* se charge, une fois que vous avez tapé le début

du nom d'une commande ou d'un alias, de terminer automatiquement le nom auquel vous faites référence. Ceci est réalisé en appuyant sur `Tab` avant la fin de votre commande. La complétion est aussi lancée sur les fichiers du répertoire courant, ainsi que sur un certain nombre de paramètres (à vous de les trouver et de les modifier).

Définir et exporter des variables

`bash`, en plus de modifier les variables d'environnement ci-dessus (plus celles qui n'ont pas été énumérées) vous offre la possibilité de définir vos propres variables. La syntaxe est la suivante :

```
<nom>=[<valeur>]
```

Si la `<valeur>` n'est pas spécifiée, `bash` affecte à `<nom>` la chaîne de caractères vide (`""`).

L'accès à la variable `<nom>` se fait en la faisant précéder du signe `$` :

```
fillmore> myvar=moi-même
fillmore> mavariable='who am i' # ne pas mettre d'espaces autour de = ...
fillmore> echo $myvar:$mavariable
moi-même:fillmore!prosper ttyp1 Sep 18 07:55 (nausica:0.0)
fillmore>
```

Voir la section sur les *quotes* pour les explications. On peut aussi mettre le nom de la variable entre accolades : `${mavariable}`.

Si vous désirez transmettre vos variables à un nouveau `bash`, ou plus généralement aux processus engendrés par le *shell* courant, il est nécessaire de les *exporter* :

```
fillmore> export mavariable # mavariable est transmise aux nouveaux shells
fillmore> bash
fillmore> echo $myvar:$mavariable
:fillmore!prosper ttyp1 Sep 18 07:55 (nausica:0.0) # myvar n'est pas reconnue
fillmore> exit # retour au shell precedent
fillmore> echo $myvar:$mavariable # l'environnement est intact
moi-meme:fillmore!prosper ttyp1 Sep 18 07:55 (nausica:0.0)
fillmore>
```

Toutes les valeurs affectées aux variables sont sujettes au mécanisme d'expansion que nous voyons ci-après.

Mécanisme d'expansion — substitution

Nous avons vu que le *shell* évaluait les valeurs des variables avant d'effectuer l'assignement. De manière plus générale, n'importe quel argument est évalué avant d'être passé à une commande. L'évaluation passe par ce qu'on appelle le *mécanisme d'expansion* qui consiste à remplacer une séquence d'un ou plusieurs caractères par la valeur qu'elle représente.

Il existe sept sortes d'expansion, réalisées selon l'ordre : expansion d'accolades, expansion du tilde (~), expansion des paramètres et des variables, substitution de commande, expansion arithmétique, éclatement de mots, expansion du nom de chemin (*pathname*).

Seuls l'expansion d'accolades, l'éclatement de mots et l'expansion du nom de chemin peuvent augmenter le nombre de mots.

Expansion d'accolades

Elle sert à générer des chaînes de caractères arbitraires. Ce mécanisme est similaire à l'expansion de nom de chemin, sauf que les noms de fichiers n'ont pas besoin d'exister. Exemple : `a{d,c,b}e` s'étend en `ade ace abe`. Les accolades doivent contenir au moins une virgule. Toute erreur de syntaxe laisse l'expression telle quelle. Exemple d'utilisation :

```
fillmore> mkdir rep{1,2,3}
```

Expansion du tilde (~)

Qui crée trois répertoires.

Si une expression commence avec un ~, le mot qui suit immédiatement est interprété comme un nom d'utilisateur, et le tout est étendu en son répertoire principal. Exemple : `ls ~prosper` affiche le contenu du répertoire personnel de l'utilisateur `prosper`. Si le ~ est suivi d'un /, alors le ~ est étendu en la valeur de `HOME`. Ainsi, `ls ~/usr` explore le répertoire `usr` qui se trouve dans votre répertoire principal. Vous en déduisez facilement que `cd ~` est strictement équivalent à `cd` (il existe aussi `cd ~+`). Dans le cas où apparaît la séquence `~-`, elle est remplacée par la valeur de `OLDPWD`. Du fait de l'ordre des expansions, le ~ peut apparaître dans les chemins (`PATH`, `CDPATH`).

Expansion des paramètres

Nous avons déjà vu que le signe `$` fournit la valeur de la variable dont le nom suit immédiatement. Des accolades peuvent entourer le nom de la variable, ce qui permet par exemple de taper des commandes du genre : `echo ${HOME}sweet`. Les accolades sont indispensables si le paramètre est un paramètre de position avec plus d'un chiffre. Des opérations sur les paramètres peuvent se réaliser dans les accolades :

- `${<nom1>:-<nom2>}` Si `<nom1>` n'est pas positionné ou nul, l'expansion de `<nom2>` est retournée. Sinon, la valeur de `<nom1>` est retournée.
- `${<nom1>:=<nom2>}` Si `<nom1>` n'est pas positionné ou nul, l'expansion de `<nom2>` est affectée à `<nom1>` et est retournée. Sinon, la valeur de `<nom1>` est retournée.
- `${<nom1>:?<nom2>}` Si `<nom1>` n'est pas positionné ou nul, l'expansion de `<nom2>` est affichée sur la sortie erreur standard et le *shell* termine s'il n'est pas interactif. Sinon, la valeur de `<nom1>` est retournée.
- `${<nom1>:+<nom2>}` Si `<nom1>` n'est pas positionné ou nul, rien n'est substitué. Sinon l'expansion de `<nom2>` est retournée.
- `${#<nom1>}` Fournit la longueur de la valeur de `<nom1>`.

- `${<nom1>#<nom2>} / ${<nom1>##<nom2>}` `<nom2>` est étendu afin de produire un motif comme pour l'expansion de noms de chemins. Si le motif correspond au début de la valeur de `<nom1>`, l'expansion est la valeur de `<nom1>` dans laquelle on a retiré le plus petit motif correspondant (cas `#`), ou le plus grand (cas `##`).
- `${<nom1>%<nom2>} / ${<nom1>%%<nom2>}` idem, sauf que le motif correspond à la fin de la valeur de `<nom1>`.

Substitution de Commandes

La substitution de commandes autorise la sortie d'une commande à remplacer son appel. Les retours chariots produits par la commande sont effacés. Il existe deux formes équivalentes de substitution de commandes : `$(<commande>)` ou `'<commande>'` (voir la section sur les quotes).

Expansion Arithmétique

L'expansion arithmétique consiste à effectuer le calcul arithmétique spécifié et à lui substituer son résultat. La syntaxe est `$((<expression>))` ou `$[<expression>]` (les crochets font partie de la syntaxe). Exemple :

```
fillmore> echo $[1+2]  
3
```

Eclatement de Mots

Les substitutions arithmétiques peuvent s'imbriquer. Le *shell* parcourt les résultats des expansions de paramètres, des substitutions de commandes et expansions arithmétiques qui ne sont pas apparus entre des guillemets (voir section sur les quotes). Les éclatements produits par ces expansions forment un nouvel éclatement qui doit répondre aux normes des séparateurs de la variable `IFS`. Le tout est donc re-plongé dans un nouvel éclatement de mots dont les entrées nulles ont été retirées.

Expansion de Nom de Chemins

Après l'éclatement des mots, **bash** regarde si chacun des mots produits contient les caractères `*`, `?` et `[`. Si c'est le cas, le mot est alors vu comme un *motif* et est remplacé par une liste triée alphabétiquement des noms de chemins correspondant au motif. La signification de ces caractères est la suivante :

- * n'importe quelle chaîne de caractères, y compris la chaîne vide.

- ? n'importe quel caractère isolé.

- [...] n'importe quel caractère apparaissant dans les crochets. [`<c1>`-`<c2>`] signifie n'importe quel caractère compris entre les caractères `<c1>` et `<c2>`. Si le premier caractère est `!` ou `^`, cela signifie n'importe quel caractère qui n'apparaît pas dans la liste. Les caractères `-` et `]` peuvent être inclus si on les positionne juste après le `[`.

Quotes

Les apostrophes(`'`, *quote*), les anti-quotes(```), les guillemets(`"`, *double quote*) et le caractère d'échappement(`\`, *escape character*) servent à retirer le côté spécial de certains caractères de **bash** ou de certains mots ayant une signification précise, ou encore à étendre leur signification.

Ainsi, chacun des caractères spéciaux vus précédemment peut retrouver son statut de

caractère normal en le faisant précéder d'un \ :

```
fillmore> echo \$  
$  
fillmore> echo \\ \& \| \" \  
\ & | " '  
fillmore>
```

L'apostrophe joue le même rôle, mais permet de regrouper les éléments (sauf lui-même) :

```
fillmore> echo '\ & | " $'  
\ & | " $  
fillmore>
```

Les guillemets servent à regrouper les éléments pour n'en former qu'un et étendent les variables. Ils autorisent aussi la déspecialisation offerte par \ :

```
fillmore> echo "Utilisateur '$UID' : \"$USER\""  
Utilisateur '7011' : prosper  
fillmore>
```

Les anti-quotes, quant à elles, autorisent l'exécution de commandes passées en arguments d'une commande :

```
fillmore> dirname 'pwd' # donne la reference absolue du rep. pere  
/home/ens/prosper/usr/tools
```

Les variables sont aussi étendues.

6.2 Forme d'une expression régulière

Une *expression régulière* est un mécanisme utilisé par un grand nombre d'outils Unix pour trouver et manipuler des motifs dans un texte. `bash` s'en sert dans son expansion de noms.

Il existe deux sortes d'expressions régulières : les expressions régulières basiques et les expressions régulières étendues. Nous nous intéressons dans un premier temps à la première.

Caractères ordinaires

Un caractère ordinaire est une expression qui se correspond elle-même. Il peut être n'importe quel caractère exceptés `<newline>` et un des caractères spéciaux mentionnés plus bas. Un caractère ordinaire précédé d'un \ est équivalent à lui-même sauf pour les caractères () { } et les chiffres.

Caractères spéciaux

Un caractère spécial précédé d'un \ est une expression régulière qui correspond au caractère lui-même. S'il n'est pas précédé d'un \, il conserve son aspect spécial :

.	[\	le point, le crochet gauche et le backslash sont spéciaux exceptés lorsqu'ils sont inclus dans une expression entre crochets.
*		l'astérisque est spéciale excepté lorsqu'elle est utilisée dans une expression entre crochets, en tant que premier caractère d'une expression, ou comme premier caractère suivant la paire \(.
^		le chapeau est spécial quand il est utilisé en premier caractère d'une expression régulière complète, ou comme premier caractère d'une expression entre crochets.
\$		le dollar est special lorsqu'il est utilisé comme dernier caractère d'une expression régulière complète.
<i>délimiteur</i>		un délimiteur d'expression régulière est spécial.

Un point, utilisé en dehors d'une expression entre crochets est une expression régulière qui correspond à n'importe quel caractère, excepté `<newline>`.

Expression entre crochets

Une expression entre crochets est une expression régulière qui correspond à un regroupement contenu dans un ensemble de regroupements représenté par l'expression entre crochets.

Une expression entre crochets est soit une liste de correspondance, soit une liste de non-correspondance et consiste en une ou plusieurs expressions dans n'importe quel ordre. Les expressions sont au choix des regroupements, des symboles de regroupement, des caractères de non-regroupement, des classes d'équivalence, des expressions d'intervalles, ou des classes de caractères.

Le crochet droit `]` perd son aspect spécial à l'intérieur d'une expression entre crochets s'il apparaît en premier dans la liste (après un `^`, s'il y en a). Sinon, il termine l'expression entre crochets.

Les caractères spéciaux `.*[\` perdent leur aspect spécial à l'intérieur d'une expression entre crochets.

Les séquences de caractères `[.`, `[=`, `[:` sont spéciales dans une expression entre crochets et sont utilisées pour délimiter les symboles de regroupement, les expressions de classes d'équivalences et les expressions de classes de caractères.

Ces symboles doivent être suivis par une expression valide, ainsi que leur délimiteur correspondant (`.]`, `=]`, `:]`).

Une **liste de correspondance** spécifie une liste qui reconnaît n'importe quel caractère qu'elle contient. Le premier caractère de la liste ne doit pas être un chapeau `^`.

Une **liste de non-correspondance** commence toujours avec un chapeau `^` et est une liste qui reconnaît n'importe quel caractère exceptés `<newline>` et les caractères de la liste. Exemple : `[^abc]` reconnaît tous les caractères, sauf `a`, `b`, `c`, et `<newline>`. Le chapeau a ce sens uniquement quand il suit immédiatement un crochet gauche.

Un **regroupement** est une séquence d'un ou plusieurs caractères qui représente un seul élément.

Un **symbole de regroupement** est un élément délimité par [`. .`]. Par exemple, [`.ch.`] correspond à la chaîne de caractères `ch` (ne pas confondre avec `ch` qui reconnaît les caractères `c` et `h`).

Un **caractère de non-regroupement** est un caractère qui est ignoré pour le regroupement. Par définition, de tels caractères ne peuvent contribuer aux expressions de classes d'équivalence ou d'intervalles.

Une expression de **classe d'équivalence** représente un ensemble de regroupements appartenant à une classe d'équivalence. Elle s'exprime en délimitant un des regroupement de la classe entre [`= =`]. Par exemple, si `a`, `A` et `A` sont dans la même classe, alors [`=a=`]`b` est équivalent à [`=à=`]`b`, .., est équivalent à [`aàAb`].

Une **expression d'intervalle** représente un ensemble de regroupements qui appartiennent à l'intervalle compris entre les deux éléments extrêmes séparés par un tiret `-`. Les extrémités doivent être valides : ce sont soit des regroupements, soit des symboles de regroupement, soit des classes d'équivalence, et la deuxième doit être supérieure à la première. Ainsi, [`a-z`] est valide, au même titre que [`=C=-e`]. Par ailleurs, les intervalles peuvent être enchaînés en rajoutant des tirets entre eux. Par exemple, [`a-m-o`] est compris comme [`a-mm-o`]. Le tiret perd son aspect spécial s'il apparaît en premier ou en dernier dans la liste ([`--@`] est donc valide).

Une **expression de classe de caractères** représente l'ensemble des caractères appartenant à la classe de caractères. Elle est délimitée par [`: :`]. Les expressions de classes de caractères sont les suivantes :

[<code>:alpha:</code>]	les lettres
[<code>:upper:</code>]	les majuscules
[<code>:lower:</code>]	les minuscules
[<code>:digit:</code>]	les chiffres
[<code>:xdigit:</code>]	les chiffres hexadécimaux
[<code>:alnum:</code>]	les lettres et les chiffres
[<code>:space:</code>]	les caractères qui affichent des blancs
[<code>:print:</code>]	les caractères d'affichage
[<code>:punct:</code>]	les caractères de ponctuation
[<code>:graph:</code>]	les caractères visibles
[<code>:blank:</code>]	les caractères blancs

Notons les expressions régulières RE.

Expressions régulières correspondant à plusieurs caractères

RE RE	La concaténation de deux RE est une RE qui correspond à la première concaténation des chaînes correspondant à chaque RE. Par exemple, la RE <code>de</code> correspond aux quatrième et cinquième caractères de la chaîne <code>abcdeabcde</code> .
RE*	l'expression RE* correspond à une chaîne contenant autant de fois que possible la RE (éventuellement 0 fois), et donc à chaque fois la première chaîne de caractères correspondant à RE. Exemple : <code>a*b</code> est identifié dans la chaîne <code>babab</code> au premier caractère <code>b</code> .
\(RE\)	Une sous-expression peut être définie à l'intérieur d'une RE en la délimitant par <code>\(\)</code> . La sous-expression a les mêmes propriétés que sa RE correspondante, sauf qu'on ne peut mettre une astérisque après <code>\)</code> .
\<chiffre>	L'expression <code>\<chiffre></code> correspond à la même RE que celle délimitée par <code>\(\)</code> . Le <code><chiffre></code> spécifie de quelle sous-expression on parle. Ceci évite de taper plusieurs fois les mêmes RE. Par exemple, dans la RE <code>\(a\)b*\([ca]\)\1\2</code> , <code>\1</code> est équivalent à <code>a</code> et <code>\2</code> est équivalent à <code>[ca]</code> . <code>\<chiffre>*</code> signifie 0 ou plusieurs occurrences de la sous-expression à laquelle <code><chiffre></code> se réfère.
RE\{<n1>,<n2>\}	Cette expression régulière correspond à un intervalle d'occurrences de RE (entre <code><n1></code> et <code><n2></code> , qui sont tous les deux compris entre 0 et 255). <code>RE\{<n1>}</code> signifie exactement <code><n1></code> occurrences de RE et <code>RE\{<n1>,\}</code> signifie au moins <code><n1></code> occurrences de RE.

Position des expressions régulières sur une ligne

On peut demander la reconnaissance de chaînes de caractères n'apparaissant qu'en début et/ou en fin de ligne.

`^RE` spécifie que RE doit correspondre à une chaîne de caractères en début de ligne. `RE$` spécifie que RE doit correspondre à une chaîne de caractères en fin de ligne. Par exemple, `^ab[ab]*c$` spécifie que la ligne doit commencer par `ab`, contenir éventuellement des `a` ou des `b` et doit se terminer par un `c`.

Expressions régulières étendues

Les expressions régulières étendues s'appliquent aux outils logiciels qui utilisent leurs notations. Nous donnons les quelques différences par rapport aux expressions régulières de base.

Nous notons les expressions régulières étendues par ERE.

- Les ERE sur les caractères isolés sont les mêmes.
- L'ensemble des caractères spéciaux est `. [\ () * + ? $ | .`

- Les délimiteurs sont définis comme tous les caractères qui permettent de délimiter une ERE.
- Les ERE entre crochets sont les mêmes que les RE.
- La reconnaissance de plusieurs caractères utilise deux ERE supplémentaires qui sont RE+ —dénote au moins une occurrence de RE, RE? —dénote zéro ou une occurrence de RE, RE|RE —dénote une alternative entre deux RE.

Les parenthèses () sont utilisées à des fins de regroupement des ERE. Par exemple :

`((ab)|c)d`

correspond aux deux chaînes de caractères `abd` et `cd`.

6.3 Commandes utilisant les expressions régulières

Quelques commandes utilisant les expressions régulières : `find`, `grep`, `emacs`, `sed`, `tr`, `man`, `less`.

6.4 Exercices

bash

► **Exercice 1.** *Modifiez le bon fichier d'initialisation afin que tous les fichiers que vous créez ne soient lisibles et modifiables que par vous mêmes, exécutables pour tout le monde.*

► **Exercice 2.** *Dans un répertoire vide, créer les fichiers `a.c`, `b.c`, `c.h`, `ab.c`, `a.cpp`, `b.cc`, puis*

1. *afficher la liste des fichiers finissant par `c`.*
2. *afficher la liste des fichiers de 3 caractères.*
3. *afficher la liste des fichiers ayant un `.` en 2ième position.*
4. *afficher la liste des fichiers ayant un `.` ou un `p` en avant dernière position.*

► **Exercice 3.** *Afficher la liste des fichiers commençant par un point et se trouvant dans votre répertoire principal.*

► **Exercice 4.** *En utilisant le fichier `/etc/passwd` afficher la liste des utilisateurs (`man cut`). (Sous l'environnement actuel avec les *yellow pages*, ce fichier possède une copie accessible : `/home/model.count/passwd.simple`).*

► **Exercice 5.** *En utilisant le fichier `/etc/passwd` afficher la liste des utilisateurs qui ont un `a` en 2ième position, puis un `a` ou `b` ou `c` ou `d` ou `e` ou ou `k` en première position (cf `cut`).*

► **Exercice 6.** *Modifier votre prompt.*

► **Exercice 7.** *Mettre dans une variable T le chemin du répertoire principal de votre chargé de TD (en moins de 10 caractères).*

Commandes Utilisant les Expressions Régulières

◆ ► **Exercice 8.** *Pour chacune des commandes listées qui utilisent des expressions régulières, taper man <commande> et essayez de voir si vous réussissez à les faire fonctionner.*

◆ ► **Exercice 9.** *Modifiez le bon fichier d'initialisation afin que tous vos fichiers core soient effacés à chaque fois que vous vous logez.*

◆ ► **Exercice 10.** *Ecrivez une commande private qui positionne récursivement les droits de tous les fichiers et les sous-répertoires “en dessous” du répertoire courant uniquement en lecture/écriture pour vous. Faites évidemment en sorte de pouvoir traverser les répertoires...*

6.5 Programmation en bash

Nous nous donnons à présent les moyens de produire des programmes en **bash**.

6.6 Scripts et paramètres

Scripts — écriture de programmes

Un *script shell* est un fichier dans lequel s'enchaînent des commandes disponibles pour le *shell*. C'est donc un fichier texte et il faut spécifier à Unix quel type de *shell* utiliser pour interpréter ce fichier.

Ainsi, tout *script bash* commence-t-il par la ligne suivante :

```
#!/bin/bash
```

où l'on peut éventuellement ajouter des options à la commande **bash**.

Comme un script est un fichier texte, Unix ne voudra pas l'exécuter si les droits en exécution ne sont pas positionnés.

A la création d'un script *shell*, ayez deux réflexes :
insérez la ligne `#!/bin/bash` en début de fichier ;
positionnez les droits en exécution sur le fichier.

Voyons à quoi ressemble un programme :

```
fillmore> cat prgm1
#!/bin/bash
#####
# Fichier: prgm1                #
# Auteur : anonyme...          #
# Contenu: Mon premier programme. #
#####
titre="Mon Premier Programme"
question="Vous plait-il ? [o/n]"
reponse="n"

echo "Ceci est $titre"
while [ "$reponse" != "o" ]
do
    echo $question;
    read reponse
done;
echo Merci
fillmore> prgm1      # lancement du programme...
Ceci est Mon Premier Programme
Vous plait-il ? [o/n]
...
```

On pourrait presque dire que la programmation en **bash** se résume à ce programme... Néanmoins, quelques subtilités restent à découvrir, comme par exemple le passage des arguments de la ligne de commande sous forme de paramètres à un script **bash**.

Un paramètre est une entité qui stocke des valeurs, un peu comme une variable dans les langages de programmation habituels. Il peut être un nom, un entier ou n'importe lequel des paramètres spéciaux définis ci-dessous. Pour un *shell* une variable est un paramètre. Les paramètres sont donc soumis au mécanisme d'affectation décrit plus haut.

Paramètres de position

Un *paramètre de position* est un paramètre dénoté par un ou plusieurs chiffres, autres que 0 tout seul. Les paramètres de position reçoivent les valeurs des arguments du *shell* lorsqu'il est invoqué et peuvent ensuite être réassignés par la commande interne **set**. Les paramètres de position sont temporairement remplacés lors de l'exécution d'une fonction *shell*.

Ainsi \$1 correspond au premier argument du script, \$2 au second, etc. Un paramètre de position contenant plus d'un chiffre doit être entouré d'accolades.

Les arguments d'un script ont pour valeurs respectives \$1, \$2, etc.

Enfin, notez l'utilisation de la commande `shift` à propos de ces paramètres (section Commandes Internes de `bash`).

Paramètres spéciaux

Plusieurs paramètres sont traités par le *shell* de manière spéciale et ne peuvent se faire assigner.

Ces paramètres sont :

- * s'étend aux paramètres de position, en démarrant à un. Le résultat de "\$*" est la suite des paramètres de position séparés par des blancs (séparateurs IFS) dans une seule chaîne.
- @ s'étend aux paramètres de position, en démarrant à un. Le résultat de "\$@" est la suite des paramètres étendus séparément chacun sous forme de chaîne.
- # s'étend en le nombre de paramètres de position (nombre décimal).
- ? s'étend en le statut du *pipeline* le plus récent lancé en avant-plan.
- s'étend en les options spécifiées à l'invocation, par la commande `set` ou par le *shell* lui-même.
- \$ s'étend en le PID du *shell*.
- ! s'étend en le PID du processus le plus récent lancé en arrière-plan.
- 0 s'étend en le nom du script *shell* ou du *shell*. Si `bash` est invoqué avec un fichier de commandes (script), \$0 est positionné au nom du fichier. Si `bash` est lancé avec l'option `-c`, \$0 est positionné au premier argument après la chaîne à exécuter. Sinon, \$0 est positionné au chemin d'accès du `bash`.
- s'étend en le dernier argument de la commande précédente, après expansion. Aussi positionné au chemin d'accès complet de chaque commande exécutée et placé dans l'environnement exporté vers cette commande.

6.7 Structures de contrôle

Les structures de contrôle sont tout simplement les boucles, les conditionnelles, ainsi que les structures qui traitent du cas par cas.

Nous donnons juste la syntaxe de ces structures.

La boucle `for`

```
for <nom> [in <mot>;]
do
  <liste>;
done
```

La liste de mots suivant `in` est étendue, ce qui génère une liste d'éléments. La variable `<nom>` est positionnée à chacun des éléments produits et `<liste>` est exécutée pour chacun d'entre eux. Si `<mot>` est omis, `<liste>` est exécutée pour chaque paramètre de position.

L'instruction `select`

```
select <nom> [in <mot>;]
do
    <liste>;
done
%
```

La liste de mots suivant `in` est étendue, ce qui génère une liste d'éléments. L'ensemble des mots étendus est affiché sur la sortie erreur standard, chacun précédé d'un nombre. Si `<mot>` est omis, les paramètres de position sont affichés. Le *prompt* PS3 est alors affiché et une ligne est lue sur l'entrée standard. Si la ligne consiste en l'un des nombres affichés, `<mot>` est positionné à ce nombre. Si la ligne est vide, l'ensemble de mots et le *prompt* sont à nouveau affichés. Si EOF est lu, la commande termine. N'importe quelle autre valeur positionne `<nom>` à vide. La ligne lue est sauvegardée dans la variable `REPLY`. La `<liste>` est exécutée à chaque sélection jusqu'à la commande `break` ou `return`. Le code de retour d'un `select` est celui de la dernière commande de `<liste>`, ou 0 si aucune commande n'a été exécutée.

L'instruction `case`

```
case <mot> in
    [<motif1> [|<motif2> ..|<motifn>])
    <liste>;;
    ..
esac
```

`<mot>` est tout d'abord étendu, puis est comparé à chaque `<motif>` tour à tour. Lorsqu'une correspondance apparaît, la `<liste>` associée est exécutée. Après la première correspondance trouvée, la comparaison se termine. Le code de retour est 0 si aucun motif ne correspond, le code retour de la dernière commande exécutée dans `<liste>` sinon.

L'instruction `if`

```
if <liste>
then
    <liste>
[elif <liste>
then
    <liste>]
```

```
..
..
[else
  <liste>]
fi
```

La <liste> du `if` est exécutée. Si le code de retour est 0, la <liste> du `then` est exécutée, sinon celle du `elif` (on recommence) s'il est présent ou sinon celle du `else` s'il est présent. Le code retour est celui de la dernière commande exécutée, ou 0 si aucune condition testée n'est vraie.

La boucle `while`

```
while <liste>
do
  <liste>
done
```

La commande `while` exécute la <liste> du `do` aussi longtemps que sa propre <liste> renvoie un code égal à 0.

La boucle `until`

```
until <liste>
do
  <liste>
done
```

La commande `until` réalise la même chose aussi longtemps que sa propre liste renvoie un code différent de 0. Le code de retour est celui de la <liste> du `do`, ou 0 si la condition n'est pas vérifiée.

6.8 Les fonctions

Voici la syntaxe d'une fonction en `bash` :

```
[function] <nom>()
{
  <liste>;
}
```

Cette commande permet de définir une fonction de nom <nom>. Le corps de la fonction est la <liste> de commandes entre accolades. Cette liste est exécutée à chaque fois que `nom` est donné sous forme de commande simple. Le code de retour de la fonction est celui de la dernière commande de <liste>.

6.9 Exercices

► Exercice 11.

1. Ecrire une commande `inf` qui boucle indéfiniment.
2. Lancer la commande `inf` en tâche de fond, et utiliser les commandes `ps` et `kill` pour l'interrompre.
3. Ecrire une commande qui détruit la dernière commande lancée en tâche de fond.

► **Exercice 12.** Ecrire une commande `args` qui affiche ses paramètres de position au moyen d'un `for <nom> do <liste> done` (for sans condition) avec ou sans la commande `shift`.



► **Exercice 13.** Ecrire une commande `rp` qui affiche ses paramètres de rang pair.
Ex :

```
fillmore> rp 1 2 3 A B C
2 A C
```

► **Exercice 14.** Ecrire une commande `tf` qui indique pour chacun de ses paramètres si c'est un fichier, un répertoire ou autre chose.

► **Exercice 15.** Ecrire une commande, ne faisant appel qu'à des fonctionnalités internes du `bash`, qui calcule l'expression numérique donnée en argument.



► **Exercice 16.** Ecrire une commande `smallwho` qui analyse le résultat de la commande `who` et affiche juste une ligne par utilisateur.

6.10 Programmation avancée

Consultez votre chargé de stage pour des exemples élaborés de programmes et essayez de réaliser ceux qui suivent.

6.11 Exercices

► Exercice 17.

1. Ecrire une commande `shell quest` qui itère une question jusqu'à la lecture au clavier d'une chaîne oui ou non.
2. Réécrire la commande `quest` pour qu'elle prenne en argument la question et qu'elle renvoie vrai ou faux suivant la réponse.

► **Exercice 18.** *Ecrire une commande `datef` qui transforme la date fournie par la commande `date` en une date exprimée en français.*

► **Exercice 19.** *Ecrire une commande qui affiche par ordre alphabétique la liste des fichiers lisibles par n'importe qui et se trouvant sous le répertoire courant.*

► **Exercice 20.** *Réaliser la commande `quiquoi` qui indique pour chaque utilisateur, la liste des programmes qu'il est en train d'exécuter.*



► **Exercice 21.** *Ecrire la commande `pstree` qui affiche l'arborescence des processus.*



► **Exercice 22.** *Ecrire une commande qui affiche les 100 premiers nombres premiers.*

Références

- [CRR96] Debra Cameron, Bill Rosenblatt, and Eric Raymond. *Learning GNU Emacs, 2nd Edition*. O'Reilly, 2nd edition, Septembre 1996. Un livre très simple d'introduction à Emacs. Existe aussi en Français. Pour plus de détails, Emacs, comme tous les logiciels GNU, est documenté sous Emacs en hypertexte (sous Emacs faire CTRL-h-i).
- [Man] *man*. Documentation des commandes, appels systèmes, formats de fichiers, ... sous Unix ; Pour avoir de l'aide sur `man` lui même : `man man` !!
- [Phi91] Jacques Philipp. *Unix : les mécanismes internes, notions de bases*. Presses de l'école nationale des ponts et chaussées, 1991. Ce livre présente les principaux algorithmes nécessaires à la mise en œuvre d'Unix.
- [Pro97] Vincent Prosper. *Stage Unix : Rappels de cours et exercices choisis*. Université de Marne-la-Vallée, 1997. Des notes de stage Unix pour les licences d'informatique. La section sur le bash de ce document est un coupé-collé de "Stage Unix : Rappels de cours et exercices choisis".
- [Rif] Jean-Marie Rifflet. *La programmation sous Unix*. Ediscience, 3ième édition. Un livre de référence pour les étudiants en informatique. Unix y est présenté en détails du point de vue du programmeur. Les appels systèmes et leur utilisation y sont soigneusement décrits.
- [Tan89] Andrew Tanenbaum. *Les systèmes d'exploitation, conception et mise en œuvre*. InterEditions, 1989. Un excellent livre sur la conception des systèmes d'exploitation, décrite à travers la conception de Minix, un mini-Unix, dont les sources sont gratuitement disponibles à la fin du livre, sur support magnétique, ou sur le web. Tout les principes sont expliqués, et pour plus de détails, on peut mettre le nez dans les sources ! L'auteur a eu de nombreux prix pour la qualité pédagogique de ses livres.

Référence rapide d'Emacs

On trouve ce document, ainsi que d'autres du même genre, sur www.refcards.com. Le copyright à la fin de la référence rapide ne concerne que la référence rapide.

GNU Emacs Reference Card

(for version 20)

Starting Emacs

To enter GNU Emacs 20, just type its name : `emacs`

To read in a file to edit, see Files, below.

Leaving Emacs

suspend Emacs (or iconify it under X)	<code>C-z</code>
exit Emacs permanently	<code>C-x C-c</code>

Files

read a file into Emacs	<code>C-x C-f</code>
save a file back to disk	<code>C-x C-s</code>
save all files	<code>C-x s</code>
insert contents of another file into this buffer	<code>C-x i</code>
replace this file with the file you really want	<code>C-x C-v</code>
write buffer to a specified file	<code>C-x C-w</code>
version control checkin/checkout	<code>C-x C-q</code>

Getting Help

The help system is simple. Type `C-h` (or `F1`) and follow the directions. If you are a first-time user, type `C-h t` for a **tutorial**.

remove help window	<code>C-x 1</code>
scroll help window	<code>C-M-v</code>
apropos : show commands matching a string	<code>C-h a</code>
show the function a key runs	<code>C-h c</code>
describe a function	<code>C-h f</code>
get mode-specific information	<code>C-h m</code>

Error Recovery

abort partially typed or executing command	<code>C-g</code>
recover a file lost by a system crash	<code>M-x recover-file</code>
undo an unwanted change	<code>C-x u</code> or <code>C-_</code>
restore a buffer to its original contents	<code>M-x revert-buffer</code>
redraw garbaged screen	<code>C-l</code>

Incremental Search

search forward	C-s
search backward	C-r
regular expression search	C-M-s
reverse regular expression search	C-M-r
select previous search string	M-p
select next later search string	M-n
exit incremental search	RET
undo effect of last character	DEL
abort current search	C-g

Use C-s or C-r again to repeat the search in either direction. If Emacs is still searching, C-g cancels only the part not done.

© 1997 Free Software Foundation, Inc. Permissions on back. v2.2

Motion

entity to move over	backward	forward
character	C-b	C-f
word	M-b	M-f
line	C-p	C-n
go to line beginning (or end)	C-a	C-e
sentence	M-a	M-e
paragraph	M-{	M-}
page	C-x [C-x]
sexp	C-M-b	C-M-f
function	C-M-a	C-M-e
go to buffer beginning (or end)	M-<	M->
scroll to next screen		C-v
scroll to previous screen		M-v
scroll left		C-x <
scroll right		C-x >
scroll current line to center of screen		C-u C-l

Killing and Deleting

entity to kill	backward	forward
character (delete, not kill)	DEL	C-d
word	M-DEL	M-d
line (to end of)	M-O C-k	C-k
sentence	C-x DEL	M-k
sexp	M-- C-M-k	C-M-k
kill region		C-w
copy region to kill ring		M-w
kill through next occurrence of <i>char</i>		M-z <i>char</i>
yank back last thing killed		C-y
replace last yank with previous kill		M-y

Marking

set mark here	C-@ or C-SPC
exchange point and mark	C-x C-x
set mark <i>arg</i> words away	M-@
mark paragraph	M-h
mark page	C-x C-p
mark sexp	C-M-@
mark function	C-M-h
mark entire buffer	C-x h

Query Replace

interactively replace a text string	M-%
using regular expressions	M-x query-replace-regexp

Valid responses in query-replace mode are

replace this one, go on to next	SPC
replace this one, don't move	,
skip to next without replacing	DEL
replace all remaining matches	!
back up to the previous match	^
exit query-replace	RET
enter recursive edit (C-M-c to exit)	C-r

Multiple Windows

When two commands are shown, the second is for “other frame.”

delete all other windows		C-x 1	
split window, above and below	C-x 2		C-x 5 2
delete this window	C-x 0		C-x 5 0
split window, side by side		C-x 3	
scroll other window		C-M-v	
switch cursor to another window	C-x o		C-x 5 o
select buffer in other window	C-x 4 b		C-x 5 b
display buffer in other window	C-x 4 C-o		C-x 5 C-o
find file in other window	C-x 4 f		C-x 5 f
find file read-only in other window	C-x 4 r		C-x 5 r
run Dired in other window	C-x 4 d		C-x 5 d
find tag in other window	C-x 4 .		C-x 5 .
grow window taller		C-x ^	
shrink window narrower		C-x {	
grow window wider		C-x }	

Formatting

indent current line (mode-dependent)	TAB
indent region (mode-dependent)	C-M-\
indent sexp (mode-dependent)	C-M-q
indent region rigidly <i>arg</i> columns	C-x TAB
insert newline after point	C-o
move rest of line vertically down	C-M-o
delete blank lines around point	C-x C-o
join line with previous (with <i>arg</i> , next)	M-^
delete all white space around point	M-\
put exactly one space at point	M-SPC
fill paragraph	M-q
set fill column	C-x f
set prefix each line starts with	C-x .
set face	M-g

Case Change

uppercase word	M-u
lowercase word	M-l
capitalize word	M-c
uppercase region	C-x C-u
lowercase region	C-x C-l

The Minibuffer

The following keys are defined in the minibuffer.

complete as much as possible	TAB
complete up to one word	SPC
complete and execute	RET
show possible completions	?
fetch previous minibuffer input	M-p
fetch later minibuffer input or default	M-n
regexp search backward through history	M-r
regexp search forward through history	M-s
abort command	C-g

Type C-x ESC ESC to edit and repeat the last command that used the minibuffer. Type F10 to activate the menu bar using the minibuffer.

GNU Emacs Reference Card

Buffers

select another buffer	C-x b
list all buffers	C-x C-b
kill a buffer	C-x k

Transposing

transpose characters	C-t
transpose words	M-t
transpose lines	C-x C-t
transpose sexps	C-M-t

Spelling Check

check spelling of current word	M-\$
check spelling of all words in region	M-x ispell-region
check spelling of entire buffer	M-x ispell-buffer

Tags

find a tag (a definition)	M-.
find next occurrence of tag	C-u M-.
specify a new tags file	M-x visit-tags-table
regexp search on all files in tags table	M-x tags-search
run query-replace on all the files	M-x tags-query-replace
continue last tags search or query-replace	M-,

Shells

execute a shell command	M-!
run a shell command on the region	M-
filter region through a shell command	C-u M-
start a shell in window <i>*shell*</i>	M-x shell

Rectangles

copy rectangle to register	C-x r r
kill rectangle	C-x r k
yank rectangle	C-x r y
open rectangle, shifting text right	C-x r o
blank out rectangle	C-x r c
prefix each line with a string	C-x r t

Abbrevs

add global abbrev	C-x a g
add mode-local abbrev	C-x a l
add global expansion for this abbrev	C-x a i g
add mode-local expansion for this abbrev	C-x a i l
explicitly expand abbrev	C-x a e
expand previous word dynamically	M-/

Regular Expressions

any single character except a newline	.	(dot)
zero or more repeats	*	
one or more repeats	+	
zero or one repeat	?	
quote regular expression special character <i>c</i>	\c	
alternative (“or”)		
grouping	(...)	
same text as <i>n</i> th group	\n	
at word break	\b	
not at word break	\B	

entity	match start	match end
line	^	\$
word	\<	\>
buffer	\‘	\’
class of characters	match these	match others
explicit set	[...]	[^ ...]
word-syntax character	\w	\W
character with syntax <i>c</i>	\sc	\Sc

International Character Sets

specify principal language	<code>M-x set-language-environment</code>
show all input methods	<code>M-x list-input-methods</code>
enable or disable input method	<code>C-\</code>
set coding system for next command	<code>C-x RET c</code>
show all coding systems	<code>M-x list-coding-systems</code>
choose preferred coding system	<code>M-x prefer-coding-system</code>

Info

enter the Info documentation reader	<code>C-h i</code>
find specified function or variable in Info	<code>C-h C-i</code>

Moving within a node :

scroll forward	<code>SPC</code>
scroll reverse	<code>DEL</code>
beginning of node	<code>.</code> (dot)

Moving between nodes :

next node	<code>n</code>
previous node	<code>p</code>
move up	<code>u</code>
select menu item by name	<code>m</code>
select <i>n</i> th menu item by number (1-9)	<code>n</code>
follow cross reference (return with 1)	<code>f</code>
return to last node you saw	<code>l</code>
return to directory node	<code>d</code>
go to any node by name	<code>g</code>

Other :

run Info tutorial	<code>h</code>
quit Info	<code>q</code>
search nodes for regexp	<code>M-s</code>

Registers

save region in register	<code>C-x r s</code>
insert register contents into buffer	<code>C-x r i</code>
save value of point in register	<code>C-x r SPC</code>
jump to point saved in register	<code>C-x r j</code>

Keyboard Macros

start defining a keyboard macro C-x (
end keyboard macro definition C-x)
execute last-defined keyboard macro C-x e
append to last keyboard macro C-u C-x (
name last keyboard macro M-x name-last-kbd-macro
insert Lisp definition in buffer M-x insert-kbd-macro

Commands Dealing with Emacs Lisp

eval **sexp** before point C-x C-e
eval current **defun** C-M-x
eval **region** M-x eval-region
read and eval minibuffer M-:
load from standard system directory M-x load-library

Simple Customization

customize variables and faces M-x customize
Making global key bindings in Emacs Lisp (examples) :
(global-set-key "\C-cg" 'goto-line)
(global-set-key "\M-#" 'query-replace-regexp)

Writing Commands

```
(defun command-name (args)  
  "documentation" (interactive "template")  
  body)
```

An example :

```
(defun this-line-to-top-of-window (line)  
  "Reposition line point is on to top of window.  
  With ARG, put point on line ARG."  
  (interactive "P")  
  (recenter (if (null line)  
                0  
                (prefix-numeric-value line))))
```

The `interactive` spec says how to read arguments interactively. Type `C-h f interactive` for more details.

Copyright © 1997 Free Software Foundation, Inc.
v2.2 for GNU Emacs version 20, June 1997
designed by Stephen Gildea

Permission is granted to make and distribute copies of this card provided the copyright notice and this permission notice are preserved on all copies.

For copies of the GNU Emacs manual, write to the Free Software Foundation, Inc.,
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA