Système d'exploitation UNIX

Linux et Solaris 2

Introduction, commandes de base et environnement de programmation

Denis Mégevand et Paul Bartholdi
Observatoire de Genève
51, chemin des Maillettes
CH-1290 Sauverny

31 octobre 2005

Table des matières

Chapitre 1
Introduction

Le système d'exploitation Unix existe pour diverses plateformes, et dans une variété de versions. Les versions les plus connues sont le système BSD 4.3 développé à Berkeley et le système V commercialisé par AT&T. Des standards essaient de se dégager de ces diverses versions.

Ce cours était à l'origine plus particulièrement basé sur les systèmes d'exploitation des stations de travail SUN, dans ses deux dernières grandes versions, la version SunOS 4.x basée sur 4.3 BSD, et SunOS 5.x — appelée aussi Solaris 2.x — basée sur SVR4 (System V Release 4) qui est une tentative d'unification des différents parfums d'Unix en leur empruntant leurs meilleures caractéristiques pour créer un système plus efficace et plus répandu.

1.1 Linux et Solaris 2

Pour cette dernière mise à jour de ces notes, nous avons abandonné la description du système SunOS 4.x au profit de Linux.

Linux est une version entièrement réécrite d'Unix, basée principalement sur le système BSD, selon les principes de l'Open Source.

À l'origine écrite et mise à disposition gratuitement au niveau source par Linus Torvald, le code en est actuellement testé et corrigé par des milliers de programmeurs de par le monde, ce qui lui assure une excellente qualité.

Les différences pouvant exister entre Linux et Solaris 2 seront mises en évidence au passage, par la signalisation des différences soit entre **Linux** et **Solaris 2**, soit entre **BSD** et **SVR4**.

La première partie (chapitres page 9 – page 27) présente quelques opérations et concepts fondamentaux à connaître pour pouvoir travailler sous Unix sur une station SUN.

Les principales commandes d'Unix sont décrites dans la seconde partie (chapitres page 29 – page

69).

Unix est un système ouvert, et son utilisation à l'intérieur d'un réseau de machines est efficace. Les principales commandes relatives au réseau sont aussi abordées au chapitre page 75, avant de voir le fonctionnement et l'utilisation du « shell » (chapitre page 79).

Quelques commandes plus complexes sont présentées en dernière partie (chapitres page 95 – page 105).

1.2 Procédure d'entrée

Pour entrer en session sur un système UNIX, il faut avoir obtenu un compte, un nom d'utilisateur, et un mot de passe. Le dialogue d'entrée est le suivant :

```
pegase login : tartempion
Password :
```

Si le mot de passe est mal tapé, le système répond :

```
Login incorrect
login :
```

et recommence le dialogue

Si le système accepte la réponse, il affiche un message d'accueil précisant la version du système utilisé, et donnant souvent quelques indications utiles.

Puis il affiche une chaîne d'invite (prompt), qui dépend du shell. Pour le TC-shell, celle-ci est modifiable. A l'Université de Genève, elle affiche par défaut la machine, le dernier élément du répertoire courant et le numéro de séquence de la commande :

```
pegase:~ 1)
```

1.3 Mot de passe

Le mot de passe de chaque utilisateur est encrypté dans un fichier, où sont également inscrits les noms des utilisateurs, les répertoires de base et d'autres données.

Le fichier `/etc/passwd` est lisible par tous, et chacun peut y écrire par l'intermédiaire des programmes `passwd`, `yppasswd` (voir page 11), `nispasswd` (voir page 11), mais pas directement.
Ce fichier décrit les utilisateurs, ligne par ligne, selon la structure suivante :

```
tartempion :DbDXItHO6QwI :4774 :4700 :TARTEMPION Jules :/home/tartempion :/bin/csh
```

A partir de Solaris 2, le fichier `/etc/passwd` ne contient plus le mot de passe, celui-ci étant mis (toujours encrypté) dans le fichier `/etc/shadow` qui n'est plus lisible que par `root`.

1.3.1 Modification du mot de passe

Pour changer son mot de passe sur un système UNIX, on utilise la commande `passwd`.

Si les NIS (voir page 11) sont installées, il faut utiliser :

```
pegase:~ 1)yppasswd
Changing NIS password for tartempion
Old password :
New password :
Retype new password :
```

Si on fait une erreur, le système donne un message.

```
Mismatch - password unchanged.
```

Si les NIS+ (voir page 11) sont installées, il faut utiliser :

```
pegase:~ 2) nispasswd
Changing password for tartempion on NIS+ server
Old login password :
New login password :
Re-enter new password :
NIS+ password information changed for tartempion
NIS+ credential information changed for tartempion
```

Si on fait une erreur, le système la signale et l'on peut la corriger.

1.4 Network Information Service : NIS

Le service NIS, anciennement appelé *Yellow Pages*, permet de gérer un réseau de systèmes UNIX divisé en *domaines*.

Chaque domaine est contrôlé par un *server*, qui contient les tables du domaine, notamment : `passwd` table des mots de passe ; `group` table des groupes d'utilisateurs ; `netgroup` table des groupes d'accès aux machines ; `ethers` table des numéros ethernet des *clients* du domaine ; `hosts` table des noms et numéros internet des machines.

A l'Observatoire, les NIS sont utilisées sous Linux, le domaine NIS est `obs.unige.ch`, grâce à la compatibilité NIS du serveur NIS+ (voir plus bas).

Pour voir une table NIS :

`ypcat` *table*

Pour voir une entrée dans une table NIS :

`ypmatch` *entree table*

1.5 NIS+ : Système d'administration centralisée amélioré

Le service NIS+ implémenté sous Solaris 2 améliore les NIS en hiérarchisant les domaines ainsi qu'au niveau de la sécurité et de l'administration. Pour l'utilisateur, seuls les noms des commandes changent. Les commandes préfixées par `yp` dans les NIS sont préfixées par `nis` dans les NIS+.

Chaque domaine ou sous-domaine est contrôlé par un *server* et des répliques (replica servers) le remplaçant lorsqu'il est inaccessible qui contiennent les tables du sous-domaine.

A l'Université, le domaine principal est `unige.ch.` et les sous-domaines sont nommés `seinf.unige.ch.`, `obs.unige.ch.`, etc.

Pour lister les tables d'un domaine NIS+ :

`nisls [-ld]` *domaine*

Pour voir une table NIS+ :

`niscat` *table*

Pour voir une entrée dans une table NIS+ :

`nismatch` *entree table*

Exercices du chapitre 1 :

1. Affichez l'adresse internet de votre machine à l'aide de la table `hosts`.
2. Trouvez votre numéro de groupe dans la table `passwd`.

Chapitre 2
Environnements de travail

Unix en soi n'offre pas un environnement de travail très convivial à l'utilisateur. Les commandes sont entrées en ligne dans une console. Cependant toutes les variantes d'Unix offrent des systèmes de fenêtrage qui permettent de travailler beaucoup plus confortablement, dans un environnement comparable à celui d'un PC ou d'un MacIntosh, avec des avantages supplémentaires.

La différence entre ces environnements réside dans l'apparence des fenêtres, dans la manière de les gérer (déplacement, modification de taille, ouverture/fermeture), dans les menus et le bureau (outil permettant de gérer des programmes ou des réglages de l'environnement.

2.1 X11

Ces environnements sont tous basés sur X11, qui est un protocole permettant d'afficher sur un écran des fenêtres tournant sur la machine pilotant l'écran ou sur une autre machine supportant le protocole X11.

Lorsque l'on ouvre sur son écran une fenêtre sur une autre machine, la machine d'affichage est appelée serveur X11, la machine où tourne le processus est le client X11.

Le serveur X11 doit autoriser le client à se connecter. Ces autorisations sont gérées par la commande `xhost`.

`xhost` affiche la liste des autorisations.
`xhost +`*machine1 machine2 ...* y ajoute les *machines*.
`xhost -` vide cette liste.

Le client doit déclarer sur quel serveur doit se faire l'affichage. Celui-ci est précisé par la variable d'environnement `DISPLAY`, qui prend la forme *machine* `:0`.

2.2 Fenêtre de login

Pour entrer en session sur un SUN ou un PC Linux de l'Observatoire, vous avez une fenêtre au milieu de l'écran vous demandant votre nom d'utilisateur et votre mot de passe. Lorsque ceux-ci ont été correctement introduits, on entre directement dans l'environnement de travail choisi.

2.3 Les fenêtres

Les fenêtres ont un *look* différent d'un environnement à l'autre. Certains environnements permettent même de varier le look des fenêtres dans l'environnement.

Une fenêtre peut être dans plusieurs états :

- **Fermée :** Elle est représentée par une icône.
- **Ouverte inactive :** Son titre et son cadre prennent une couleur ou une apparence assez terne. Les environnements jouent en général sur les impressions de relief pour montrer l'activité d'une fenêtre.
- **Ouverte active :** Son titre est représenté sur un fond en relief ou de couleur plus vive.
- **Enroulée :** Dans KDE et Gnome. Le corps de la fenêtre disparaît, alors que la barre de titre reste, comme si on avait enroulé un écran de projection dans son boitier.

2.3.1 La souris

La souris est un élément important, qui pilote le curseur et permet de gérer les fonctions du système.

Les environnements définissent les propriétés et le comportement de la souris, comportement qui peut être modifié par une application.

Les boutons de la souris ont des fonctions différentes dans les différents environnements, et dans les différentes applications, bien que la tendance soit à une standardisation des fonctions :

- **Gauche :** Positionnement et sélection.
- **Milieu :** Divers.
- **Droit :** Menus.

2.3.2 Positionnement et sélections

Les boutons de gauche et parfois du milieu permettent, dans une fenêtre, de positionner le curseur et de sélectionner du texte, pour le détruire, le déplacer ou le copier.

On clicke sur le bouton de gauche une fois pour positionner le curseur à un emplacement de la fenêtre.

On clicke sur le bouton de gauche, éventuellement plusieurs fois, et on maintient le bouton appuyé pour effectuer une sélection. Les possibilités de sélections dépendent des applications, mais obéissent souvent au standard suivant :

- **1 click :** Sélection par caractères
- **2 clicks :** Sélection par mots
- **3 clicks :** Sélection par lignes.
- **4 clicks :** Sélection de tout le texte de la fenêtre.

2.3.3 Les menus

Le bouton de droite donne habituellement accès à des menus.

Lorsque le curseur est dans le fond, le menu principal permet d'accéder à des programmes et utilitaires importants du système.

Dans une fenêtre, on accède au menu spécifique à l'application.

Sur le cadre ou dans une icône, on accède au menu de gestion de la fenêtre.

Pour activer une option de menu, il faut déplacer la souris tout en maintenant le bouton enfoncé. Les flèches donnent accès aux sous-menus.

2.3.4 Gestion des fenêtres avec la souris

On peut ouvrir une fenêtre en appuyant deux fois rapidement (double-click) sur le bouton de gauche après avoir placé le curseur dans une icône.

On peut déplacer une icône avec le bouton de gauche.

On active une fenêtre ouverte soit en cliquant dans la fenêtre, soit en y placant le curseur. Le comportement peut être réglé au gré de l'utilisateur.

On peut accéder directement à certaines fonctions de gestion en placant le curseur sur le cadre d'une fenêtre ouverte. La encore, le comportement dépend fortement de l'environnement utilisé et on se reportera utilement au manuel de celui-ci pour plus de détails.

2.3.5 Les ascenseurs

Certaines fenêtres peuvent mémoriser plus d'informations qu'elles ne peuvent en afficher. Dans ce cas, un ascenseur permettant de controller la position de la partie affichée est placé à droite ou à gauche de la fenêtre. Sur certaines fenêtres on peut activer ou non l'ascenseur et régler la position de celui par rapport à la fenêtre.

On peut faire défiler les fenêtres possédant un ascenseur vertical ou horizontal de plusieurs manières :

Les triangles aux deux bouts permettent en cliquant de déplacer la fenêtre par lignes. La partie de l'ascenseur mise en évidence représente le pourcentage affiché de la fenêtre. Pour se déplacer :

- **Par ligne :** cliquer sur les flèches.
- **Par page :** cliquer sur le cable en dessous ou au dessus des flèches.
- **Au début ou à la fin :** cliquer sur les ancres.
- **De manière quelconque :** cliquer le carré entre les flèches et déplacer l'ascenseur sans relâcher le bouton de gauche de la souris.

2.4 Facilités du clavier SUN sous Solaris

Des touches de fonctions à gauche du clavier permettent d'accélerer les opérations dans l'environnement de fenêtres.

Les touches `COPY`, `PASTE`, et `CUT` permettent de transférer du texte d'une fenêtre à l'autre, ou à l'intérieur d'une fenêtre.

La touche `HELP` donne une fenêtre d'aide dans certains cas.

Les touches `OPEN` et `FRONT` sont des raccourcis pour la gestion des fenêtres.

Les touches `AGAIN`, `UNDO`, et `FIND` sont utiles dans les fenêtres textes.

Chapitre 3
Système de Fichiers

Le fichier est l'unité de base du système UNIX.

Sous UNIX, pratiquement tout est traité comme un fichier.

Un fichier peut représenter :
- un ensemble de données
- un programme exécutable
- un ensemble de commandes (script)
- un périphérique
- un répertoire
- rien (`/dev/null` est un « trou noir »)

Un fichier UNIX n'a qu'un nom, et pas d'extension. Son nom peut contenir 255[1] caractères, y compris des caractères spéciaux. Pratiquement tous les caractères peuvent être contenus dans un nom de fichier. Les caractères spéciaux du shell `! & ( ) ' " ‘` de même que les espaces doivent être protégés par un backslash lorsqu'on les écrit, et par conséquent ne sont pas recommandés pour les noms de fichiers.

UNIX différencie les majuscules des minuscules.

```
pegase:~ 3) touch Ceci\ est\ le\ fichier_au_nom_le_plus_bizarre_qu\'il_m\'ait-
_jamais_ete_donne_de_creer\(sous_UNIX\)!
```

donne un fichier

```
Ceci est le fichier_au_nom_le_plus_bizarre_qu'il_m'ait_jamais_ete_donne_de-
_creer(sous_UNIX)!
```

Un système de fichiers UNIX est une structure hiérarchisée de fichiers.

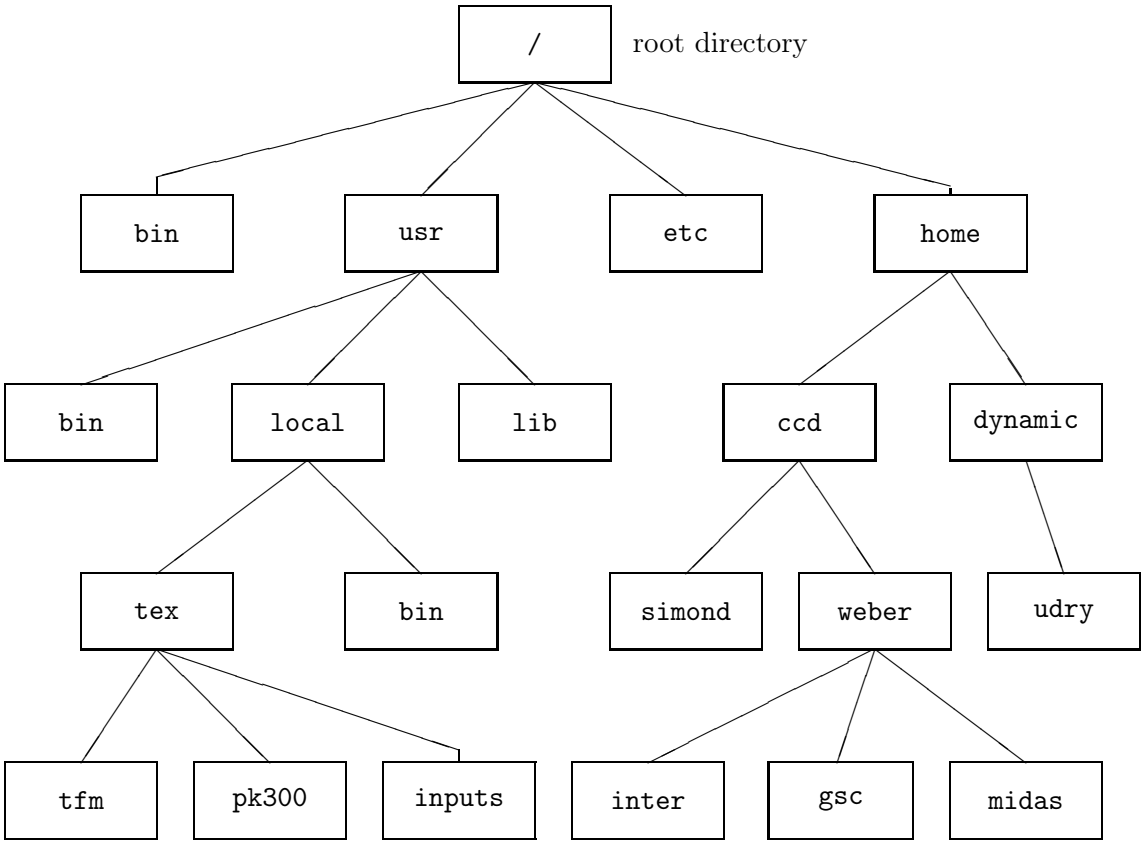
[1] Certaines version d'Unix limitent cette longueur à 16 ou 32 caractères ! On aura avantage à ne pas dépasser 16 caractères, si l'on désire pouvoir porter son application sur d'autre machines

Certains de ces fichiers, appelés répertoires, peuvent contenir d'autres fichiers, y compris des répertoires.

Les répertoires sont les branches d'un arbre, alors que les fichiers d'un autre type en sont les feuilles.

On représente habituellement cet arbre de manière inversée.

3.1 Structure hiérarchisée de fichiers



3.2 Chemins d'accès

On peut accéder à un fichier se trouvant quelque part dans le système de manière absolue ou de manière relative.

Un chemin d'accès commençant par un slash / est *absolu*. Il représente la position du fichier dans le sytème de fichiers complet de la machine.

Un chemin d'accès relatif commence par un composant. La recherche se fait alors depuis le répertoire courant.

3.2.1 Composants de chemins d'accès

Un chemin d'accès est constitué de composants séparés par des /

– *nom de répertoire ou de fichier*
– abbréviations :

`/`	le répertoire racine (le plus haut)
`.`	le répertoire courant
`..`	le répertoire parent (au dessus)
`~`	le répertoire de base de l'utilisateur « home directory »
`~jo`	le répertoire de base de l'utilisateur *jo*

3.3 UFS : Disques et partitions

Chaque disque contient un certain nombre de partitions logiques, qui sont montés comme des systèmes de fichiers BSD (UFS) sous Solaris 2 ou étendus (ext2 ou ext3) sous Linux.

Le nom des disques et des partitions sont différents en système **Linux** et **Solaris 2** :

`/dev/sd`*nx* **Linux**

avec *n* n° du disque dépendant de l'adresse SCSI.
x lettre (a–h) représentant la partition.

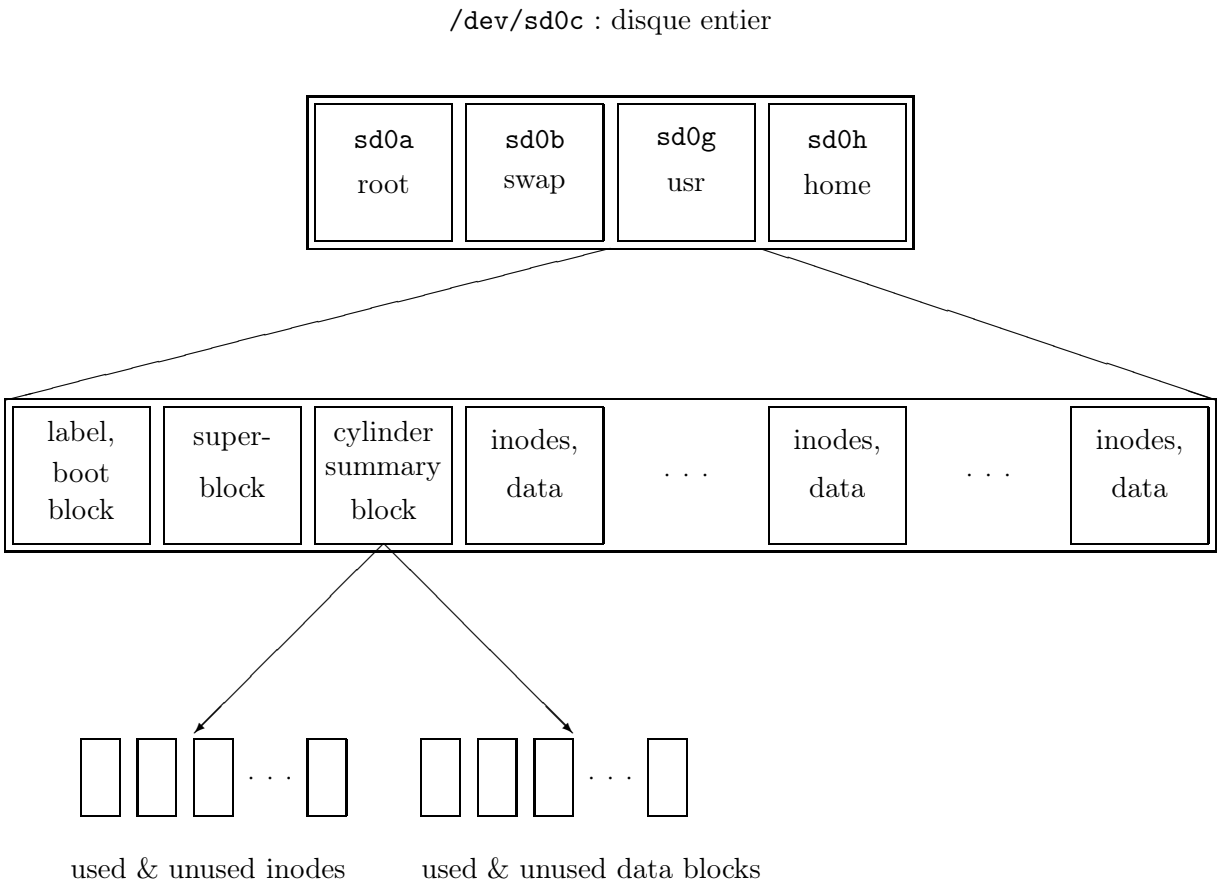
`/dev/dsk/c`*c*`[t`*t*`]d`*d*`s`*s* **Solaris 2**

avec *c* n° du controlleur du bus (SCSI,PCI) ou du disque.
t adresse sur le bus.
d n° du disque (0 lorsque le disque est sur un bus).
s n° de la partition (0–7).

Une partition par machine est réservée pour l'espace de swapping, dans les autres sont construits des *systèmes de fichiers*.

Chaque système de fichiers est formé d'un certain nombre de groupe de *cylindres*. Les cylindres contiennent des structures de contrôle, des inodes et des blocs de données.

Chaque système de fichier contient les blocs de *bootstrap*, un *super-bloc*, les blocs de *récapitulation des cylindres*, et les blocs *d'inodes* et de *données*.



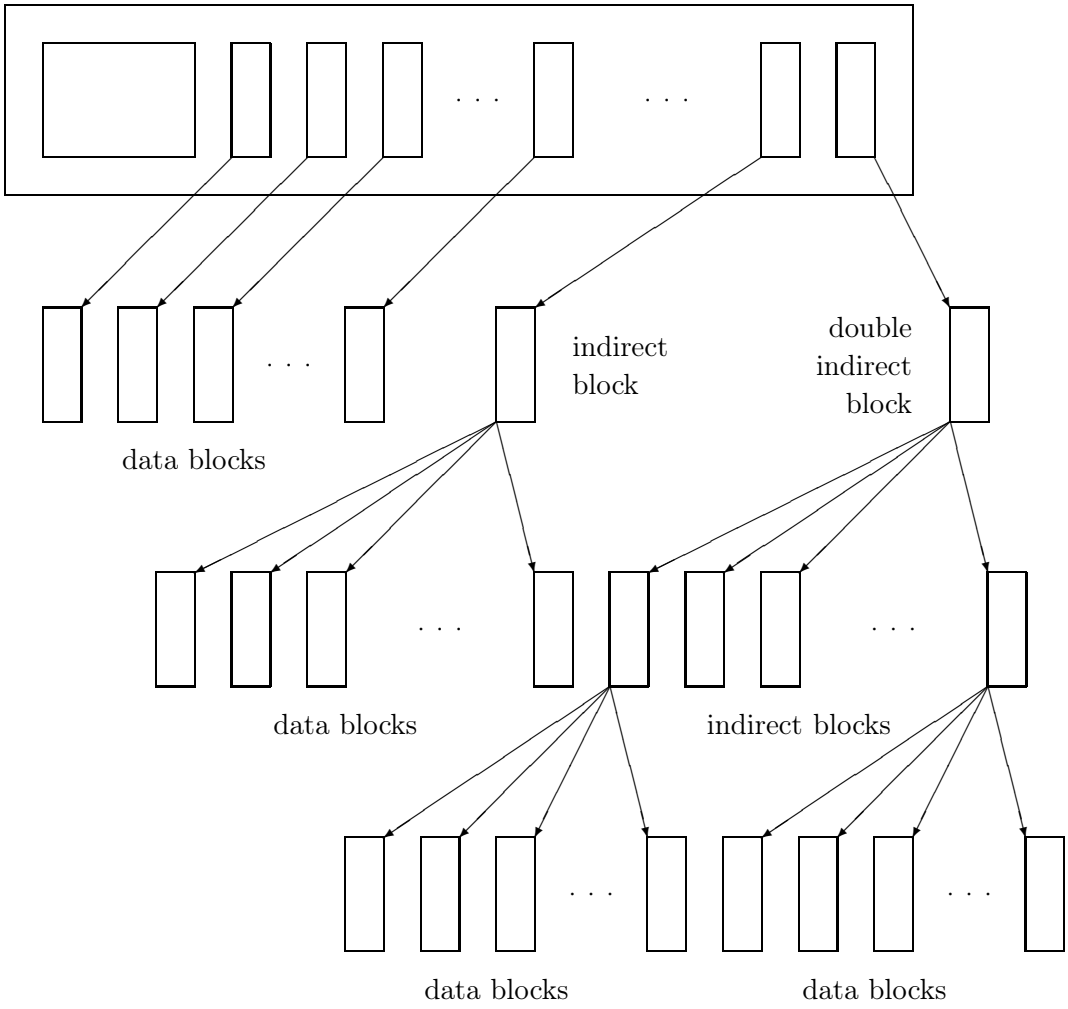
3.4 Super-bloc et inodes

Chaque système de fichiers est décrit par son *super-bloc*, qui liste les groupes de cylindres, la taille du système, le nombre d'inodes et le nombre de blocs de données.

Le super-bloc est recopié dans chaque groupe de cylindre par sécurité.

Un inode est un enregistrement structuré décrivant un fichier.

Elle contient des informations sur le types du fichier, son propriétaire, ses restrictions d'accès, les adresses des blocs de données et des blocs d'indirections, etc ...



3.5 Types de fichiers sous UNIX

Il existe 7 types de fichiers sous UNIX.

d	des répertoires
b	des fichiers spéciaux de type bloc
c	des fichiers spéciaux de type caractère
l	des liens symboliques
p	des fichiers spéciaux de type FIFO (First In First Out)
s	des sockets (ports virtuels)
-	des fichiers ordinaires

3.5.1 Répertoire

Un répertoire est un fichier ordinaire dans le sens où il possède un descripteur et des données sous forme de suite d'octets.

Seul le système peut écrire dans un répertoire. Un répertoire est une table de couples (i-node, chaîne de caractères). Le i-node est le numéro du descripteur de fichier. Ce couple est appelé lien.

3.5.2 Fichiers

Les caractéristiques associées à un fichier (la date de sa création, la date de la dernière modification, la taille, le propriétaire etc...) sont regroupées dans un descripteur de fichier (inode).

Le système UNIX permet à un seul fichier physique d'avoir plusieurs noms, grâce aux liens.

Le deuxième champ de la commande `ls -l` permet de voir le nombre de liens que possède un fichier.

3.5.3 Périphériques

A chaque périphérique est associé un ou plusieurs fichiers spéciaux de type bloc ou caractère.

Un fichier spécial possède un descripteur de fichier, comme un fichier ordinaire, mais pas de données.

Ceci permet à l'utilisateur de voir les périphériques comme des fichiers, et donc d'utiliser une seule syntaxe pour les commandes.

Un fichier spécial est caractérisé de facon unique par son type (bloc ou caractère), son majeur (type de périphérique) et son mineur (une instance de ce périphérique).

Pour créer un fichier de périphérique :

`mknod` *fichier* `[c|b]` *major minor*

Seul le *superuser* peut utiliser cette commande. Il est recommandé d'utiliser `makedev` pour installer les périphériques habituels.

3.5.4 Liens

Il existe deux types de liens : les liens rigides « *hards* » et les liens symboliques.

Le *lien rigide* est une nouvelle entrée du répertoire, identique à celle du fichier. Le i-node est le même, les deux fichiers sont indifférentiables.

Le *lien symbolique* est un fichier spécial qui contient le nom du fichier sur lequel il pointe. Ce dernier peut très bien ne pas exister.

Pour créer un lien :

`ln [-s]` *fichier* `[`*lien*`]`

Si le *lien* est absent, celui-ci prend le *fichier*, et est créé dans le repertoire courant.

Exemple :

```
pegase:~ 4) cd /usr/local
pegase:~ 5) ln -s /import/lib
pegase:~ 6) ls -l lib
lrwxrwxrwx  1 root          11 Dec  5 10:46 lib -> /import/lib/
pegase:~ 7) ln -s ‘date +%Y%m%d‘ Done
pegase:~ 8) ls -l Done
lrwxrwxrwx  1 root           8 Oct 24 14:46 Done -> 20011024
```

3.5.5 Tubes nommés

Un fichier spécial de type FIFO permet à deux processus d'échanger des données (pipe).

Un processus écrit dans le fichier tandis que l'autre lit. Les données ne sont jamais stockées sur le disque.

Les processus doivent être executés de manière asynchrones.(&)

Pour créer un fichier spécial de type FIFO :

`/etc/mknod` *fichier* `p`

Exemple : Le compilateur C traite toujours un fichier. Un tube nommé permet de compiler un fichier compressé sans créer de version source décompressée, donc sans utiliser de place disque :

```
pegase:~ 7) /etc/mknod tampon.c p
pegase:~ 8) zcat bigprog.c.Z > tampon.c &  ; cc -o bigprog tampon.c
[1] 3012

[1]  Done     zcat bigprog.c.Z > tampon.c
```

L'exécutable `bigprog` est créé.

3.6 `uid` et `gid`

Un numéro d'utilisateur (`uid`) et de groupe (`gid`) est associé à chaque personne travaillant sur un système UNIX.

Ces numéros sont initialisés lors du `login` en consultant le fichier `/etc/passwd`.

Au milieu d'une session de travail, il est possible de changer momentanément son identité, c'est-à-dire son `uid` en utilisant la commande `su` :

`su [-] [`*username*`]`

Par défaut, le *username* est celui de *root*, c'est-à-dire l'utilisateur privilégié, qui gère le système (superuser).L'option `-` permet d'exécuter une procédure complète de login. Dans tous les cas il faut connaitre le mot de passe de la personne dont on prend l'identité.

3.7 Propriété d'un fichier

Chaque fichier appartient à un propriétaire (*user*) et à un groupe (*group*). Ceux-ci sont définis en utilisant le `uid` et le `gid` du processus créant le fichier.

Pour changer le propriétaire et le groupe, voir les commandes `chown` et `chgrp` décrites aux sections page 44 et page 44.

3.8 Accès aux fichiers

Les autorisations d'accès aux fichiers sont définies pour le propriétaire (*user*), le groupe (*group*) et les autres (*others*). Les accès possibles sont : *lecture*, *écriture* et *exécution*.

Pour un répertoire, la permission d'exécution permet d'effectuer une recherche dans le répertoire.

`rwx`	`rwx`	`rwx`
u*ser*	g*roup*	o*ther*

Il existe quatre autres permissions importantes : `setuid`, `setgid` *mandatory locking* et *sticky bit.*

3.8.1 `setuid`, `setgid` et *sticky bit*

Pour un exécutable :

– le `setuid` permet à un processus d'être executé sous le `uid` du propriétaire du fichier.
– le `setgid` permet à un processus d'être executé sous le groupe propriétaire du fichier (`gid`).
– le *sticky bit* demande au système d'exploitation de garder une image du programme en mémoire.

Pour un fichier accessible en lecture-écriture :

– le `mandatory locking` permet de bloquer l'accès à un fichier lorsqu'il est accédé par un programme. Cette faculté de blocage obligatoire n'est disponible que sous **SVR4**.

Pour un répertoire :

– le `setgid` ordonne à un fichier d'utiliser le groupe du parent. **BSD**
– le *sticky bit* : seul le propriétaire du fichier peut le détruire. **BSD**

3.8.2 `chmod`

Pour changer l'accès aux fichiers :

`chmod` *mode fichier ...*

Le *mode* peut être donné en octal en additionnant les permissions suivantes :

– Permissions spéciales

4000	Set user id (`setuid`)
2000	Set group id (`setgid`)
1000	*Sticky bit*

– Permissions habituelles

400	Lecture pour le propriétaire
200	Écriture pour le propriétaire
100	Exécution pour le propriétaire
40	Lecture pour le groupe
20	Écriture pour le groupe
10	Exécution pour le groupe
4	Lecture pour les autres
2	Écriture pour les autres
1	Exécution pour les autres

Le *mode* peut aussi être mis de manière symbolique :

`chmod [who]` *op perm* `[,who` *op perm*`]...` *fichier ...*

– l'argument `who` est une combinaison de :

u	Utilisateur (propriétaire individuel du fichier)
g	Groupe possédant le fichier
o	Autres utilisateurs
a	Tous les usagers du systeme = ugo (defaut de who).

– l'argument *op* représente les opérations que doit effectuer `chmod` :

+ Ajouter les autorisations spécifiées ;

− Oter les autorisations spécifiées

= Initialiser les autorisations spécifiées

– l'argument *perm* est une combinaison de :

r Lecture

w Écriture

x Exécution

l mandatory locking **SVR4**

s setuid ou setgid

t sticky bit

Exemples :

```
pegase:~ 9) chmod 755 my_program
pegase:~ 10) chmod a+x,g-w,o-w my_script
```

3.9 umask

La commande `umask` permet de modifier le *masque* des droits d'accès à la création de fichier.

`umask` *masque*

Lorsqu'un programme crée un fichier, il spécifie les droits d'accès. Parmi ceux-ci, certains sont accordés, d'autres refusés, en fonction du masque.

Le masque est donné en octal. Les bits valant 1 correspondent à des droits refusés.

Exemple :

```
pegase:~ 11) umask 022
```

Si *emacs* crée un fichier, il utilisera comme droits d'accès `rw-rw-rw- (666)`, en appliquant le masque `----w--w- (022)`, on obtiendra comme permissions : `rw-r--r-- (644)`.

3.10 NFS : Systèmes de fichiers importés

NFS est un système qui permet de partager des systèmes de fichiers à travers le réseau.

On peut grâce à lui accéder à des fichiers sur les disques d'autres machines que celle sur laquelle on travaille.

Il existe deux manières d'accéder aux disques situés sur une autre machine :

– par un point de montage (`/unige/`, `/obs/` etc.)

– via `/net/`*machine*`/export/`*disque*`/...`

La première présente l'avantage que l'on n'a pas à se préoccuper de savoir sur quelle machine se trouvent les fichiers qui nous intéressent.

La seconde d'accéder à des fichiers spécifiques sur une autre machine, sans qu'ils apparaissent nécessairement dans une table de montage.

3.11 Répertoire `unsaved`

La plupart des utilisateurs ont accès par défaut à deux partitions dont l'une (d'environ 1 GB) est régulièrement sauvée et l'autre (généralement beaucoup plus grande) ne l'est jamais automatiquement. La seconde est montée dans le répertoire `unsaved` du répertoire de base (`home`).

Exercices du chapitre 3 :

1. Donnez trois manières différentes d'accéder à votre répertoire de base.
2. Pourquoi `cd /..` ne donne-t-il pas de message d'erreur ?
3. Sur quel disque se trouve la partition `unsaved` ?

Chapitre 4

Filtres et Redirections

La notion de filtre est très importante sous UNIX. Les auteurs d'UNIX, Ritchie et Kernighan, ont remarqué que souvent les fichiers de données étaient traités séquentiellement par plusieurs programmes, avant de donner les résultats définitifs.

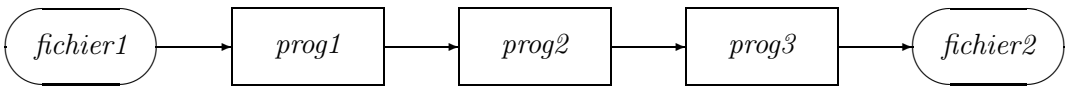
Ils ont donc developpé la plupart des outils UNIX comme des filtres traitant un flux de données fourni par l'entrée standard, puis envoyant les résultats à la sortie standard.

L'entrée et la sortie standard sont par défaut le clavier et l'écran de la machine ou du terminal où l'on travaille. Mais on peut les rediriger, à l'aide des symbôles suivants :

`>` redirige la sortie standard sur un fichier.

`<` redirige un fichier sur l'entrée standard.

`|` redirige la sortie standard vers l'entrée standard d'un autre programme (tube).



Les données du *fichier1* devant être successivement traitées et modifiées par le programmes *prog1*, *prog2* et *prog3* et mises ensuite dans *fichier2* seront traitées par la ligne de commande :

prog1 < *fichier1* | *prog2* | *prog3* > *fichier2*

Voir au chapitre `Shell` (page 79), pour plus d'informations concernant les redirections, notamment pour les messages d'erreur et les différences selon le type de shell utilisé.

Exemples :

```
pegase:~ 12) echo "essai 1" > test
pegase:~ 13) ls | more
```

Exercices du chapitre 4 :

Que font les opérations :

1. `echo "hello, world"`
2. `echo "hello, world" > test`
3. `ls /usr/lib`
4. `ls /usr/lib | more`
5. `ls /usr/lib > more`
6. `more test`
7. `chmod 755 more`
8. `more test`

Chapitre 5
Commandes de base

Les sections suivantes présentent les commandes principales d'UNIX.

Chaque commande est présentée par un titre de sous-section donnant le nom de la commande, une courte description de son action.

Les lignes suivantes montrent les différentes formes qu'elle peut prendre, avec les options principales. Ces lignes sont données en caractères `teletype` pour les parties fixes, et en *italique* pour les mots-clés à remplacer par des noms réels. Trois points de suspension signifient que l'argument précédent peut être multiple.

D'éventuelles remarques suivent, avant la description des options.

Certaines commandes plus complexes sont décrites de manière différente.

Les syntaxes, options ou descriptions spécifiques au système Linux (Berkeley) ou Solaris 2.x (Système V R4) sont mises en évidence par les sigles **BSD** et **SVR4** placés en fin de ligne. Si les descriptions sont trop différentes pour être traitées de cette manière, des sous-sections décrivent de manière spécifique chaque version.

Chapitre 6
Informations génerales

6.1 man

Affiche le manuel d'une commande à l'écran.

`man [-s` *section*`]` *titre...* **SVR4**
`man [`*section*`]` *titre...* **BSD**
`man -k` *mot-clé*

Une indication facultative de section permet de séparer les entrées homonymes. Il y a huit sections (essayer `man man`), la commande `man intro` donne le résumé de chaque section.

Options :

`-k.` Résume sur une ligne toutes les commandes relatives au *mot-clé*.

`-s` Permet de spécifier la *section* **SVR4**.

6.2 whoami

Affiche le nom de l'utilisateur.

```
whoami
```

6.3 who

Affiche les utilisateurs.

`who [`*am i*`|`*am I*`]`
`who [-q]`

`who` affiche le nom de tous les utilisateurs connectés, le nom de leur terminal et le temps de connexion. Avec les deux arguments *am i* ou *am I*, `who` ne liste que l'utilisateur appelant.

`who` avec l'option `-q` résume le nom des utilisateurs connectés et leur nombre.

6.4 `w`

Résume l'activité des utilisateurs et du système.

`w [-hls] [`*user*`]`

Affiche une entête sur le système, ainsi qu'une ligne par connexion, donnant l'utilisateur, le processus et des indications sur la session.

Options :

`-h` supprime l'entête.

`-l` affiche une ligne complète par session [défaut].

`-s` affiche une ligne réduitepar session.

`-u` n'affiche que l'entête. **SVR4**

6.5 `df`

Informe sur l'espace disque d'un système de fichier.

6.5.1 BSD

Sans l'option `-i`, l'affichage donne, en kilobytes, l'espace total, utilisé et libre, la proportion utilisée et le point de montage.

`df [-a][-h][-i][-t` *type*`][`*fs...*`]`

Si *fs* est un fichier, `df` informe sur le système de fichier contenant celui-ci.

Options :

`-a` informe aussi sur les systèmes vides ou non-montés.

`-h` donne des nombres plus «humains».

`-i` informe sur les inodes.

`-t` informe sur les systèmes d'un *type* défini.

6.5.2 SVR4

Sans options, l'affichage donne l'espace libre en blocs de 512 bytes et le nombre de fichiers disponibles.

`df [-abegknt][-F` *type*`][`*fs...*`]`

Si *fs* est un répertoire, `df` informe sur le système de fichier contenant celui-ci.

Options :

`-a` informe aussi sur les systèmes vides ou non-montés.

`-b` informe seulement sur l'espace libre.

`-e` informe seulement sur les fichiers libres.

`-g` donne une information complète.

`-k` informe en kilobytes à la manière de **BSD**.

`-n` informe seulement sur le type de système.

`-t` donne également l'information sur l'espace et le nombre de fichiers totaux.

`-F` informe sur les systèmes d'un *type* défini.

6.6 `du`

Informe sur l'espace disque utilisé.

`du [-a][-s][`*répertoire...*`]`

Donne le nombre de KB (**BSD**) ou de blocs de 512 bytes (**SVR4**) utilisés dans les répertoires spécifiés et leurs sous-répertoires.

Options :

`-a` informe aussi sur les fichiers.

`-d` informe seulement sur le système de fichier courant. **SVR4**

`-k` donne l'information en kilobytes. **SVR4**

`-o` n'ajoute pas les sous-répertoires. **SVR4**

`-s` informe seulement sur le total des répertoires.

Exercices du chapitre 6 :

1. Quelle différence y a-t-il entre les commandes `whoami` et `who am i` ?
2. Combien de kilobytes sont utilisés dans votre répertoire principal et ses sous-répertoires ?

Chapitre 7
Gestion des répertoires

Un nom de répertoire est toujours spécifié par un chemin d'accès, absolu ou relatif, tel que décrit précédemment.

7.1 `mkdir`

Crée un répertoire.

`mkdir [-p]` *nom-du-répertoire*

Options :

`-p` création des répertoires intermédiaires manquants.

7.2 `rmdir`

Detruit des répertoires.

`rmdir` *nom-du-répertoire...*

Un répertoire n'est détruit que s'il est vide (Attention aux fichiers cachés, voir page 41).

7.3 `cd`

Change de répertoire courant.

`cd [`*nom-du-répertoire*`]`

Il faut avoir la permission d'exécution pour le nouveau répertoire.

Sans paramètre, la commande `cd` ramène dans le répertoire de base.

On peut spécifier une variable `cdpath` contenant des sommets d'arbres de recherche pour `cd` (voir votre fichier `.login`).

7.4 pwd dirs

Affiche le répertoire courant.

```
pwd
dirs
```

La commande `dirs` est interne au C-shell, elle est plus rapide, mais parfois moins précise :

```
pegase:~ 14) cd ~/tex/lettres
pegase:~ 15) mv ~/tex ~/texte
pegase:~ 16) pwd
/home/system/tartempion/texte/lettres
pegase:~ 17) dirs
~/tex/lettres
```

Explication : Si l'on renomme un répertoire intermédiaire dans l'arbre de recherche, `pwd` et `dirs` donnent des résultats différents, car `pwd` parcourt le système de fichier pour voir où on est, alors que `dirs` restitue le nom enregistré au changement.

Exercices du chapitre 7 :

1. Dans quelles circonstances la commande `cd ..` ne marche-t-elle pas ?
2. Créez un sous-répertoire `cours`, et faites-en votre répertoire de travail.

Chapitre 8
Gestion des fichiers

8.1 cp

Copie des fichiers.

`cp [-ip]` *fichier1 fichier2*
`cp -r[-ip]` *répertoire1 répertoire2*
`cp [-ipr]` *fichier... répertoire*

Options :

`-i` confirmation en cas de superposition.

`-p` copie également les attributs des fichiers.

`-r` copie également les sous-répertoires des fichiers.

8.2 dd

Convertit et copie un fichier.

`dd [`*option*`=`*valeur*`]...`

Cette commande permet de copier des fichiers spéciaux comme le montre l'exemple.

Options :

`if=`*fichier* fichier d'entrée.

`of=`*fichier* fichier de sortie.

`conv=ascii` conversion ascii-ebcdic,

`conv=ebcdic` conversion ebcdic-ascii

`conv=block` enregistrement variable-fixe.

`conv=lcase` conversion en minuscules.

`conv=ucase` conversion en majuscules.

...

Exemple :

```
pegase:~ 18) dd if=/dev/fd0 | rsh obsipc2 dd of=/dev/fd0
```

8.3 mv

Déplace ou renomme des fichiers.

`mv [-fi][-|--]` *fichier1 fichier2*
`mv [-fi][-|--]` *répertoire1 répertoire2*
`mv [-fi][-|--]` *fichier... répertoire*

Options :

`-f` ignore les restrictions de mode.

`-i` confirmation en cas de superposition.

`-` les arguments suivants sont pris comme noms de fichiers (accès aux fichiers commencant par -). **BSD**

`--` les arguments suivants sont pris comme noms de fichiers (accès aux fichiers commencant par -). **SVR4**

8.4 rm

Détruit des fichiers.

`rm [-fir][-|--]` *fichier...*

Options :

`-f` ignore les restrictions de mode.

`-i` confirmation de chaque destruction.

`-r` détruit récursivement les fichiers des fichiers de type répertoires cités, ainsi que les sous-répertoires qu'ils contiennent.

`-` les arguments suivants sont pris comme noms de fichiers (accès aux fichiers commencant par -). **BSD**

`--` les arguments suivants sont pris comme noms de fichiers (accès aux fichiers commencant par -). **SVR4**

Exercices du chapitre 8 :

1. Copiez votre fichier `~/.login` dans un fichier `test1`, en conservant sa date de création originale.
2. Transformez le fichier `test1` en fichier `test2` ne contenant que des minuscules ?
3. Déplacez tout le contenu de votre répertoire `cours` dans un nouveau répertoire `cours-unix`, copiez-y votre fichier `~/.login`.

4. Détruisez votre répertoire cours-unix et son contenu en une commande. Vérifiez ensuite le nom du répertoire courant. Revenez à votre répertoire de base.

Chapitre 9
Informations sur les fichiers

9.1 `ls`

Liste les fichiers.

`ls [-aFl1]` [*fichier...*]

Options :

`-a` liste aussi les fichiers *cachés*.

`-F` repère (/ * @ =) certains types de fichiers.

`-l` liste des informations détaillées des fichiers, et le contenu des répertoires.

`-1` liste les fichiers sur une colonne.

`-R` liste les répertoires récursivement.

9.2 `file`

Détermine le type des fichiers.

`file [-m` *magicfile*`]` *fichier...*

Le début des fichiers est examiné par rapport à certaines règles pour déterminer ce qu'il contient.

Les règles sont contenues dans le fichier `/etc/magic`. Un autre fichier peut être utilisé avec l'option `-m` *magicfile*.

Exercices du chapitre 9 :

1. Sachant que le premier champ numérique d'un listing obtenu par `ls -l` représente le nombre de liens sur un fichier, trouvez combien de sous-répertoires a `/usr/local`. Cet exercice fait appel à des options de `ls` non-décrites ici, utilisez `man`.
2. Essayer d'écrire un fichier `magic` pour que `file -m magic` reconnaisse les documents LaTeX. La structure d'un fichier `magic` est décrite au début du fichier `/etc/magic`.

Chapitre 10
Modifications des fichiers

10.1 `touch`

Met à jour les dates de dernier accès et de dernière modification de fichiers.

`touch [-acm]` *fichier...* **SVR4**
`touch [-cf]` *fichier...* **BSD**

Si le *fichier* n'existe pas, il est créé.

Options :

`-a` Ne modifie que la date de dernier accès. **SVR4**

`-c` ne crée pas un fichier inexistant.

`-f` tente d'ignorer les restrictions de mode. **BSD**

`-m` Ne modifie que la date de dernière modification. **SVR4**

10.2 `chmod`

Modifie les permissions d'un fichier.

`chmod [-fR]` *mode fichier*

Options :

`-f` tente d'ignorer les restrictions de mode.

`-R` récursif dans les sous-répertoires.

La commande et son paramètre *mode* est décrite en détail à la section page 24.

10.3 `chown`

Modifie le propriétaire de fichiers.

`chown [-fhR]` *nom fichier...* **SVR4**
`chown [-fR]` *nom*`[.`*groupe*`]` *fichier...* **BSD**

Sous **BSD**, seul le *super-user* peut exécuter cette commande. En **SVR4**, le propriétaire peut également l'exécuter dans certains cas.

Options :

`-f` ne rapporte pas les erreurs.

`-h` ne suit pas les liens symboliques. **SVR4**

`-R` récursif dans les sous-répertoires.

10.4 `chgrp`

Modifie le groupe propriétaire de fichiers.

`chgrp [-fhR]` *groupe fichier...* **SVR4**
`chgrp [-fR]` *groupe fichier...* **BSD**

Sous **BSD**, seul le propriétaire appartenant aux deux groupes ou le *super-user* peut exécuter cette commande. En **SVR4**, le propriétaire peut également l'exécuter dans certains cas sans appartenir au groupe d'arrivée.

Options :

`-f` ne rapporte pas les erreurs.

`-h` ne suit pas les liens symboliques. **SVR4**

`-R` récursif dans les sous-répertoires.

10.5 `rehash`

Commande interne au C-shell qui met à jour la table de recherche des exécutables dans le *path* ou chemin de recherche.

`rehash`

Permet au shell de recalculer sa *hash-table* interne pour y inclure de nouveaux exécutables.

Exercices du chapitre 10 :

1. Créez un fichier `essai` vide, et voyez quelles sont ses attributs et protections.
2. Quelles sont les autorisations minimales pour que vous puissiez exécuter un fichier vous appartenant ? Testez cela sur le fichier `essai` récemment crée.

Chapitre 11
Affichage des fichiers

11.1 `cat`

Concatène et affiche des fichiers.

`cat [-nsv] [`*fichier...*`]`

Lit les fichiers séquentiellement, et les affiche à l'écran. Si rien n'est spécifié, lit l'entrée standard jusqu'à un « CTRL-D ».

Options :

`-n` numérote les lignes.

`-s` n'avertit pas si le fichier est inexistant. **SVR4**

`-s` raccourcit une suite de lignes blanches. **BSD**

`-e` affiche les fins de lignes.

`-t` affiche les tabulateurs.

`-v` affiche les caractères de contrôle.

11.2 `more` `less` `page`

Affiche des fichiers par écran.

`more [-ds] [`*fichier...*`]`
`less [-ds] [`*fichier...*`]`
`page [-ds] [`*fichier...*`]`

Lit les fichiers séquentiellement, et les affiche à l'écran, avec un arrêt après chaque écran. Si aucun fichier n'est spécifié, l'entrée standard est lue.

Options :

`-d` donne une erreur lors de fausses commandes, ou supprime les messages d'erreur (pour `less`).

`-s` raccourcit une suite de lignes blanches.

`page` nettoie l'écran avant d'afficher.
`more` superpose les écrans.
`less` a plus de possibilités que `more` et permet la navigation avec les flèches du clavier.

Entre chaque écran, on doit entrer une commande où *i* est facultatif :

Commandes :

i`<SPACE>` affiche un autre écran ou *i* lignes.

i`<CR>` affiche *i* lignes de plus.

i`b` recule de *i* écrans et affiche un écran.

`/`*chaîne* cherche *chaîne* et affiche un écran à partir de cette ligne.

`:p` affiche le fichier précédent.

`:n` affiche le prochain fichier.

`?` affiche un rappel des différentes commandes accessibles.

`q` termine la commande.

11.3 `head`

Affiche le début de fichiers.

`head [-`*nombre*`] [`*fichier...*`]`

Options :

-nombre spécifie le nombre de lignes à afficher. Par défaut, affiche 10 lignes.

11.4 `tail`

Affiche la fin d'un fichier.

`tail [-|+`*nombre*`] [-r] [`*fichier*`]`
`tail [-|+`*nombre*`] [-f] [`*fichier*`]`

Options :

`+`***nombre*** on commence l'affichage à *nombre* de lignes du début du fichier.

`-`***nombre*** on commence l'affichage à *nombre* de lignes de la fin du fichier (défaut `:-10`).

`-f` Si l'entrée n'est pas un tube, se met en attente à la fin du fichier et affiche les lignes éventuellement ajoutées au fichier par d'autres processus.

`-r` Affiche les lignes demandées en sens inverse.

11.5 `od`

Affiche un fichier dans un format spécifique, par bytes, mots de 16 bits (mots16), de 32 bits (mots32) ou de 64 bits (mots64).

`od [-`*format*`] [`*fichier*`]`

Le nom vient de *octal dump*, qui rappelle le format par défaut.

Options :

`-b`	par bytes, en octal non-signé.	
`-c`	par bytes, en caractères ASCII.	
`-D`	par mots32, en décimal non-signé.	**SVR4**
`-d`	par mots16, en décimal non-signé.	
`-F`	par mots64, en double flottants .	**SVR4**
`-f`	par mots32, en flottants.	
`-h`	par mots16, en hexadécimal non-signé.	**BSD**
`-O`	par mots32, en octal non-signé.	**SVR4**
`-o`	par mots16, en octal non-signé (défaut).	
`-S`	par mots32, en décimal signé.	**SVR4**
`-s`	par mots16, en décimal signé (défaut).	**SVR4**
`-s`*n*	cherche des chaînes de caractères terminées.	**BSD**
`-X`	par mots32, en hexadécimal signé.	**SVR4**
`-x`	par mots16, en hexadécimal signé (défaut).	

Exercices du chapitre 11 :

1. Utilisez `cat` pour lister sur l'écran le contenu de vos fichiers `.alias` et `.alias.local`.
2. Sachant que `man` utilise la commande `more` pour l'affichage des pages de manuel, trouvez rapidement le manuel de la commande interne `history`.

Chapitre 12
Comparaison de fichiers

12.1 `comm`

Informe sur les lignes communes de deux fichiers

`comm [-123]` *fichier1 fichier2*

Affiche les lignes uniques aux *fichiers* dans les colonnes 1 et 2, et les lignes communes en colonne 3. Toutes les combinaisons d'options peuvent être données.

Options :

-1 supprime la colonne 1 (lignes du *fichier1*).

-2 supprime la colonne 2 (lignes du *fichier2*).

-3 supprime la colonne 3.

12.2 `cmp`

Compare deux fichiers

`cmp [-l]` *fichier1 fichier2*

Affiche la ligne où les deux fichiers commencent à différer.

Options :

-l affiche les différences, byte par byte.

12.3 `diff`

Compare deux fichiers et affiche les différences

`diff [-biw][-D`*chaîne*`]` *fichier1 fichier2*
`diff [-biw][-S`*fichier*`]` *rép1 rép2*

Donne les modifications à apporter au *fichier1* pour qu'il concorde avec le *fichier2*. Si l'un des deux *fichiers* est un répertoire, `diff` compare deux fichiers de même noms, dont l'un est dans ce répertoire.

Pour deux répertoires, `diff` donne une liste des noms de fichiers diffèrents, après tri.

Options :

-b ignore les blancs en fin de ligne.

-i ignore les différences majuscules-minuscules.

-w ignore tous les blancs.

-D crée une version compactée des deux fichiers, avec des instructions pour le préprocesseur C, pour que l'option `-D`*chaîne* donne une compilation du *fichier2*, et sinon du *fichier1*.

-S commence la comparaison des répertoires au *fichier*.

Exercices du chapitre 12 :

1. Fabriquez deux fichiers tests diffférents et comparez les à l'aide des différentes commandes décrites. Essayez avec des fichiers binaires, ou avec un mélange des deux.

Chapitre 13
Autres traitements des fichiers

13.1 `wc`

Compte les caractères, mots et lignes de fichiers.

`wc [-clw][`*fichier...*`]`

`wc` affiche les comptes individuels et totaux (options par défaut : `-clw`).

Options :

-c compte les caractères.

-l compte les lignes.

-w compte les mots.

13.2 `split`

Découpe un fichier en plus petits.

`split [-`*nombre*`][`*fichier1* `[`*fichier2*`]]`

`split` lit *fichier1*, le découpe selon le *nombre* de lignes spécifié, et en fait des fichiers dont le nom *fichier2* est suffixé par *aa*, *ab*, etc...

Par défaut, *fichier2* est *x*.

13.3 `paste`

Joint les lignes de fichiers. **SVR4**

`paste [-d`*liste*`]`*fichier1 fichier2...*
`paste -s [-d`*liste*`]`*fichier*

`paste` aboute les lignes correspondantes de plusieurs *fichiers*, ou les lignes successives d'un *fichier*, avec l'option `-s`.

Les lignes sont jointes par un `TAB` ou par les caractères de la *liste*, avec l'option `-d`.

13.4 sort

Trie les lignes d'un fichier.

`sort [-bfnru] [-o` *fichier*`][-t`*car*`][+`*n1*`[-`*n2*`]] [`*fichier...*`]`

La combinaison de l'option `-t` définissant le séparateur de champ, et des options `+`*n1* et `-`*n2* précisant les champs clés (sauter *n1* champs et trier jusqu'au champ *n2*) permettent des tris très performants.

Options :

`-b` ignore les blancs en début de champ-clé.

`-f` ne tient pas compte des différences entre majuscules et minuscules.

`-n` trie numériquement.

`-o` envoie les résultats dans le *fichier*.

`-r` trie en ordre inversé.

`-t` utilise *car* comme séparateur de champ à la place des blancs.

`-u` supprime les occurences multiples des lignes considérées comme identique par les clés de tri.

`+`*n1* spécifie le nombre de champs à sauter avant le début d'une clé.

`-`*n2* précise le champ où s'arrête la clé.

Exemple :

```
pegase:~ 19) niscat passwd | sort -t : +3n -4 +0 -1f
...
```

trie les utilisateurs numériquement par groupes (quatrième champ), puis par nom d'utilisateur, en omettant les différences majuscules-minuscules sur ce champ.

13.5 uniq

Traite la multiplicité des lignes d'un fichier.

`uniq [-cdu] [+`*n1*`][-`*n2*`] [`*fichier1* `[`*fichier2*`]]`

L'entrée doit être triée. Les options `d` et `u` sont prises par défaut.

Options :

`-c` imprime chaque ligne différente une fois, avec son nombre d'occurence.

`-d` imprime une fois chaque ligne à occurence multiple.

`-u` imprime uniquement les lignes à simple occurence.

`+`*n1* spécifie le nombre de caractères à sauter avant la comparaison.

`-`*n2* spécifie le nombre de champs à sauter avant la comparaison.

13.6 `find`

Recherche les fichiers satisfaisants certains critères.

`find` *répertoire...expression*

`find` traverse les répertoires spécifiés, et leurs sous-répertoires et teste chaque fichier avec la partie expression de la ligne de commande, où l'on peut utiliser les primitives décrites ci-dessous. Le résultat de la recherche peut donner lieu à une action. Par défaut, `find` ne fait rien des fichiers trouvés.. Les caractères spéciaux du shell doivent être protégés.

Primitives :

`-name` *motif*	vrai pour les fichiers dont le nom correspond au *motif*.
`-user` *nom*	vrai pour les fichiers appartenant à *nom*.
`-group` *nom*	vrai pour les fichiers du groupe *nom*.
`-size` *n*	vrai pour les fichiers de taille *n* blocs.
`-atime` *n*	vrai pour les fichiers accédés durant les *n* derniers jours.
`-mtime` *n*	vrai pour les fichiers modifiés pour le dernière fois il y a *n* jours.
`-a`	opérateur *et* entre deux expressions.
`-o`	opérateur *ou* entre deux expressions.
`!`	opérateur *non* sur une expression.
`(...)`	groupement d'expressions primitives.
`-print`	action d'imprimer le nom des fichiers trouvés.
`-exec` *commande*`\;`	action d'exécuter la *commande* pour les fichiers trouvés, dont le nom peut être passé à la *commande* par `{}`.
`-ok` *commande*`\;`	idem, mais avec une demande de confirmation pour chaque exécution.

Exemples :

```
pegase:~ 20) find ~  ! -name \*.f -print
pegase:~ 21) find ~ \( \( -name \*% -mtime +3 \) \
   -o -name core \) -exec /bin/rm -f {} \;
```

La première commande imprime le nom de toutes les sources Fortran de l'utilisateur. La seconde détruit toutes les versions antérieures non-modifiées depuis trois jours et tous les fichiers `core`

Exercices du chapitre 13 :

1. Combien y a-t-il d'utilisateurs inscrits sur votre réseau *NIS* ou *NIS+* local ?

Chapitre 14
Informations temporelles

L'horloge interne utilisée par la machine pour toutes les informations temporelles n'est pas toujours très précise. On peut la recaler régulièrement à l'aide de programmes de synchronisation travaillant sur une horloge de référence, située dans la machine où ailleurs sur le réseau. Dans ce cas, on obtient des heures précises à quelques millisecondes.

14.1 `date`

Affiche la date et l'heure.

```
date [-u]
```

Options :

`-u` affiche l'heure GMT au lieu de MET.

14.2 `time`

Informe sur les temps CPU, système et réel.

```
time [commande]
```

Affiche le temps (utilisateur, système et réellement écoulé) pris par l'exécution d'une *commande* ou par le shell courant.

Donne aussi des indications sur la mémoire occupée, les entrées-sorties et les opérations de swapping.

14.3 `cal`

Affiche le calendrier.

`cal [[`*month*`]` *year*`]`

Affiche un calendrier de l'année spécifiée.

Si un mois est aussi donné en paramètre, affiche un calendrier seulement pour ce mois.

Affiche un calendrier du mois courant si rien n'est précisé.

Le mois est donné par un nombre entre 1 et 12, l'année par un nombre entre 1 et 9999.

Remarque : ***Le calendrier produit est anglais. Le passage du calendrier Julien au Grégorien a eu lieu pour le monde Britannique en septembre 1752, ce qui crée un décalage de 12 jours entre la sortie de*** `cal` ***et le calendrier officiel européen, d'octobre 1582 à septembre 1752.***

Chapitre 15
Impression de fichiers

La gestion des imprimantes et l'impression de fichiers est une des fonctions où les différences entre le système **BSD** et le système **V** sont les plus marquées. Nous avons adopté à l'Observatoire le système d'impression `CUPS`, souvent utilisé dans les systèmes **Linux**, qui est une version plus récente des mêmes concepts et propose les commandes spécifiques des deux systèmes. Nous recommandons toutefois d'utiliser les commandes BSD décrites ci-dessous.

Toutes les imprimantes de l'Observatoire sont connectées directement sur le réseau Ethernet, et chaque station a ses propres queues de *spooling*. Ceci a le défaut qu'il n'est pas possible d'interroger l'état global d'une queue d'impression puisque celle-ci n'existe pas. Par contre, aucune imprimante ne peut être bloquée parce qu'une autre machine est elle même arrêtée.

15.1 Système V versus BSD

En système **SVR4**, la notion de classe regroupe un ensemble d'imprimantes semblables, parmi lesquelles le système peut choisir en fonction de la charge. Aucune classe n'est pour l'instant définie à l'Observatoire. Les commandes de base sont :

`lp` pour soumettre une impression.
`lpstat` pour examiner l'état des queues d'impression et des classes.
`cancel` pour supprimer un job de la queue

En système **BSD**, les queues sont définies dans un fichier `/etc/printcap`, et les commandes de base sont :

`lpr` pour soumettre une impression.
`lpq` pour examiner l'état des queues d'impression.
`lprm` pour supprimer un job de la queue

15.2 Impression de fichier texte `lprtxt`

La commande `lprtxt` permet d'imprimer un fichier texte en défnissant la police et la taille des caractères, l'orientation du papier, et éventuellement un entête et la pagination.

`lprtxt [-i]` *fichier*

Options :

`-i` commute le mode interactif de `lprtxt`

15.3 `lpr`

Soumet un fichier à l'impression.

`lpr [-P`*imprimante*`][-#`*copies*`]` *fichier*

Options :

`-P` imprime sur une *imprimante* particulière.

`-#` spécifie le nombre de copies.

15.4 `lpq`

Affiche le status de l'imprimante par défaut.

`lpq [-P`*imprimante*`]`

Options :

`-P` affiche le status d'une *imprimante* particulière.

Notez que l'on ne peut voir que les impressions lancées à partir de la station où l'on se trouve. Il n'est pas possible de voire les impressions venant d'une autre station.

15.5 `lprm`

Annule une soumission.

`lprm [-P`*imprimante*`][`*job...*`]`

Sans option d'imprimante, annule dans la queue par défaut. Sans numéro de job, annule le job courant.

Options :

`-P` choisit la queue d'une *imprimante* particulière.

Notez qu'il n'est pas possible d'annuler une imression venant d'une autre station, même si vous en êtes propriétaire.

15.6 Nom des imprimantes

Les noms des imprimantes et des queues qui sont installées suivent une certaine logique.

L'entête se rapport au constructeur (`hp` pour Hewlett–Packard, `tek` pour Tektronix-Xerox, etc.)

Le chiffre qui suit est un numéro d'ordre d'installation.

Le suffixe indique une queue particulière (type, format ou orientation du papier) :

(*rien*)	format a4 simple face
`d`	a4 recto-verso, reliure sur le grand côté
`t`	a4 recto-verso, reliure à l'italienne
`_a3`	a3 simple face
`_a3d`	a3 recto-verso, reliure sur le grand côté
`_a3t`	a3 recto-verso, reliure à l'italienne
`_tr`	a4 transparent (pour les `tek`)
`_manual`	alimentation manuelle (pour les `tek`)

Exemple :

```
hp0_a3d   tek0_manual   tek0_tr   hp7t
```

15.7 Impression de fichiers divers

Le système d'impression `cups` permet de reconnaitre la plupart des types de fichiers (texte ascii, images (gif, jpeg, tiff etc.), postscript, pdf...avec ou sans compression, et de les imprimer directement sans que l'utilisateur doivent d'abord les convertir dans le format de l'imprimante.

Il se peut que vous tombiez une fois ou l'autre sur un type de fichier qui ne soit pas reconnu par le système. Comme il est possible de le rajouter pour en faire profiter toute la communauté, veuillez nous le signaler en nous donnant une copie du fichier.

Exemple :

```
lpr -Ptek0 M31.jpg
lpr -Php0_a3d Andromede.gif.gz
```

Chapitre 16
Commandes diverses

16.1 `echo`

Affiche ses arguments sur la sortie standard.

La version Système V et la version interne au tcsh comprennent les chaînes de caractères suivantes : **SVR4**
`\b`=backspace,
`\f`=form-feed,
`\n`=new-line,
`\r`=carriage return,
`\t`=horizontal tab,
`\v`=vertical tab,
`\\`=backslash,
`\0`*n*=caractère à huit bits dont le code ASCII en octal est *n*,
`\c`=interrompt la chaîne et supprime le `LF` à la fin de celle-ci.

`echo [-n]` *chaîne...*

Options :

`-n` n'ajoute pas de `LF` à la fin.

16.2 `tty`

Affiche le nom du terminal.

`tty [-s]`

Affiche le nom du terminal, et rend un status valant 0 si l'entrée standard est un terminal, et 1 sinon.

En Système V, le status vaut 2 si des options illégales sont utilisées. **SVR4**

Options :

`-s` n'affiche rien, mais renvoie le status.

16.3 `write` `wall` `mesg`

Gestion des messages entre utilisateurs

`write` *utilisateur*
`wall [-a] [`*fichier*`]`
`mesg [{y|n}]`

`write` envoie un message à l'*utilisateur*. Finir le message par un `CTRL-D`.
`wall` envoie un message ou un *fichier* à tous les utilisateurs connectés. L'option `-a` envoie à toutes les fenêtres.
`mesg` permet d'autoriser ou d'inhiber l'arrivée de messages d'autres utilisateurs sur le terminal.

Chapitre 17
Développement de programmes

17.1 nedit

Edite un fichier dans les fenêtres.

`nedit` [*fichier*]

C'est un éditeur très convivial est suffisamment puissant pour de nombreuses applications. L'utilisation en est simple, basée sur les sélections et la souris.

Chaque fonction est activée dans un menu, et agit sur la zone sélectionnée.

De nombreux modes spécialiśes sont activés automatiquement selon le suffixe du nom de fichier, pour developper des programmes en C, en Fortran, etc. ou pour préparer un document en TEX ou en LATEX.

17.2 crisp

Edite un fichier dans les fenêtres.

`crisp` [*fichier*]

C'est la version Unix de l'éditeur BRIEF. L'utilisation en est simple, basée sur les sélections et la souris.

Chaque fonction est activée dans un menu, et agit sur la zone sélectionnée. Une aide en ligne importante est accessible. Ce programme est puissant, rapide et peut traiter de très gros fichiers.

De nombreux modes spécialiśes sont activés automatiquement selon le suffixe du nom de fichier, pour developper des programmes en C, en Fortran, etc. ou pour préparer un document en TEX ou en LATEX.

17.3 emacs

Edite un fichier dans ou hors des fenêtres.

`emacs [`*fichier*`]`

C'est l'éditeur préférentiel sur les terminaux.

Les déplacements sont liés aux touches du curseur.

Les commandes sont liées à des caractères de contrôle (`C-`*clé*) ou à une séquence de deux caractères `ESC` et *clé* (`M-`*clé*).

L'éditeur Emacs utilisé à l'Observatoire est GNU Emacs, version 20, developpé au MIT par Richard Stallman et la Free Software Foundation. Il est du domaine public, et très bien fait.

De nombreux modes spécialiśes sont activés automatiquement selon le suffixe du nom de fichier, pour developper des programmes en C, en Fortran, etc. ou pour préparer un document en TEX ou en LATEX. Ces modes ont de nombreuses facilités et des commandes particulièrement puissantes pour écrire proprement vos sources et les tester efficacement.

17.3.1 L'écran

L'écran présente en haut une ligne de menus et en bas une ligne de status, en vidéo inversée. Le minibuffer où on entre les commandes est situé tout en bas.

On peut diviser l'écran en plusieurs fenêtres verticales avec la commande `C-x 2`.

On passe d'une fenêtre à l'autre avec la commande `C-x o`.

Pour détruire les autres fenêtres, on a la commande `C-x 1`.

17.3.2 Commandes de base

`C-x C-f`	file load	chargement d'un fichier dans le buffer de travail.
`C-x C-w`	file save	sauvetage du buffer de travail sur disque.
`C-x C-c`	exit	fin du programme et retour au système.
`C-x i`	file insert	insertion d'un fichier dans le buffer de travail.
`M-%`	query-replace	recherche et remplacement avec demande de confirmation.
`C-@`	set mark	marquage de la position du pointeur comme début de zone.
`C-w`	kill region	destruction d'une zone définie depuis la marque jusqu'au pointeur.
`C-k`	kill line	destruction d'une ligne ou partie de ligne.
`M-d`	kill word	destruction d'un mot ou partie de mot.
`C-y`	unkill	récupération de la dernière destruction.
`M-y`	unkill and pop	remplacement de la dernière récupération par la destruction précédente.
`C-g`	abort command	interruption d'une commande en cours.
`C-x u`	undo	annulation de l'effet de la dernière commande (cumulable).
`M-x`	extended command	permet d'entrer n'importe quelle commande Emacs.

17.4 cc

Compilateur C

`cc` [`-flags`][`-c`][`-o` *sortie*] *fichier*...[`-l`*libr*]... **SVR4**
`cc` [`-help`][`-c`][`-o` *sortie*] *fichier*...[`-l`*libr*]... **BSD**

Compile et édite les liens du *fichier* source C (`.c`). Par défaut, la sortie est mise dans un fichier `a.out`. Les *librairies* chargées avec l'option `-l` sont cherchées dans le chemin de recherche des librairies, et le nom court *libr* est préfixé par `lib` et suffixé par `.a`.

Options :

`-flags` liste toutes les options de `cc`. **SVR4**

--help liste toutes les options de `cc`. **BSD**

`-c` fabrique seulement le code objet.

`-l` spécifie les *librairies*.

`-o` spécifie le nom du fichier de *sortie*.

En faits, les compilateurs Sun récents (environnement *workshop*) ont un très grand nombre d'options qu'il serait difficile de décrire ici. Le meilleur moyen de les connaitre est de regarder les pages `man` ou encore mieux d'utiliser la commande `cc -xhelp=readme` (et/ou `cc -xhelp=flag` et `cc -xhelp=errors`).

17.5 lint

Vérifie la cohérence syntaxique d'un programme C.

`lint` *fichier*

Analyse de manière stricte la syntaxe d'un programme C, et donne des messages d'erreur ou des avertissements sur la non-portabilité, le gaspillage ou la non-utilisation de ressources, comme par exemple :

```
printf returns value which is always ignored
```

17.6 f77 f90

Compilateur Fortran

`f77` [`-flags`][`-c`][`-o` *sortie*] *fichier*...[`-l`*libr*]...
`f90` [`-flags`][`-c`][`-o` *sortie*] *fichier*...[`-l`*libr*]...

Compile et édite les liens du *fichier* source fortran (`.f` ou `.for`). Par défaut, la sortie est mise dans un fichier `a.out`. Les *librairies* chargées avec l'option `-l` sont cherchées dans le chemin de recherche des librairies, et le nom court *libr* est préfixé par `lib` et suffixé par `.a`.

Comme pour `cc`, on aura avantage à consulter les pages `man`, ou à utiliser les commandes `f77 -xhelp=readme` (et/ou `f77 -xhelp=flag` et `f77 -xhelp=errors`).

Options :

`-flags` liste toutes les options. **SVR4**

`-c` fabrique seulement le code objet.

`-l` spécifie les *librairies*.

`-o` spécifie le nom du fichier de *sortie* .

17.7 Options d'optimisation pour les compilateurs

L'option `-fast` est un bon compromis entre le temps de compilation et le temps d'exécution.

Les options `-xO1` à `-xO5` définissent des optimisations de plus en plus poussées du code exécutable. Les plus hauts niveaux de doivent être utilisés qu'avec circonspection, car le code généré est très sensible aux effets de bord.

Les options `-dalign` , `-xlibmil` , `-xarch=v9` ou `-xarch=v9a` sont souvent efficaces.

Inversément, l'option `-g` ne doit être utilisée que lorsque l'on veut utiliser le *debugger*, car elle ralentit beaucoup l'exécution.

17.8 ar

Crée et gère une librairie ou un fichier archive

`ar d|m|r|t|x [csuv]` *librairie* [*fichier...*] **SVR4**
`ar d|r|t|x [cuv]` *librairie* [*fichier...*] **BSD**

Gère un groupe de fichiers combinés en une librairie. Les commandes suivantes (`lorder`, `tsort`, et `ranlib`) permettent de ranger ou d'indexer une librairie pour en accélérer le chargement. Sous **SVR4**, `ar` génère automatiquement une table des symboles de la librairie, ce qui rend obsolète la commande `ranlib`.

Options :

`d`	enlève les fichiers spécifiés de la librairie.	
`m`	déplace les fichiers spécifiés à la fin de la librairie.	**SVR4**
`r`	ajoute ou remplace les fichiers spécifiés dans la librairie.	
`t`	liste le contenu de la librairie.	
`x`	extrait les fichiers spécifiés de la librairie.	
`c`	crée la librairie.	**BSD**
`c`	supprime le message si la librairie est créée..	**SVR4**
`s`	force la mise à jour de la table des symboles de la librairie.	**SVR4**
`u`	remplace seulement les fichiers modifiés.	
`v`	*verbose*, permet de controller le déroulement des opérations..	

Exemple :

```
pegase:~ 22) ar cr libplot.a arc.o putsi.o move.o box.o
```

17.9 lorder

Liste les paires de routines liées d'une librairie.

`lorder` *librairie*

Le listing produit peut être utilisé par `tsort` pour créer une librairie que `ld` peut charger en un seul passage.

```
pegase:~ 23) lorder libplot.a
```

```
box.o box.o
arc.o arc.o
move.o move.o
putsi.o putsi.o
box.o move.o
arc.o putsi.o
move.o putsi.o
```

17.10 tsort

Trie topologiquement

`tsort` *fichier*

`tsort` produit un listing totalement ordré des éléments donnés en entrée avec leurs relations d'ordre partielles.

```
pegase:~ 24) lorder libplot.a | tsort
box.o
arc.o
move.o
putsi.o
```

17.11 ranlib

Convertit une archive en librairie indexée. **BSD**

`ranlib` *librairie*

Fabrique une librairie indexée plus rapide à charger par `ld`. `lorder` n'est pas nécéssaire lorsque l'on utilise `ranlib`.

Chapitre 18
Jobs et processus

Un processus peut être exécuté dans deux modes : le mode *foreground* ou d'avant-plan bloque le shell jusqu'à la fin du processus ; le mode *background* ou d'arrière-plan permet l'exécution simultanée du processus et du shell.

On peut interrompre l'exécution d'un processus d'avant-plan par les séquences :

`<CTRL-C>` on tue le processus définitivement.
`<CTRL-Z>` on stoppe le processus, et on retourne au shell. On peut le relancer en background par la commande `bg`, ou en foreground par la commande `fg`.

18.1 ps

Informe sur les processus.

`ps [-aefl] [-u` *user*`]` **SVR4**
`ps [-ax][l|u|v]` **BSD**

Affiche une ligne par processus de l'utilisateur.

Options :

`-a` informe sur les processus des autres utilisateurs. **BSD**
`-a` n'informe pas sur les processus sans terminaux. **SVR4**
`-e` informe sur l'ensemble des processus actifs. **SVR4**
`-f` affiche des informations orientées utilisateurs. **SVR4**
`-l` affiche des informations orientées processus.
`-u` *user* affiche tous les processus d'un utilisateur. **SVR4**
`-u` affiche des informations orientées utilisateurs. **BSD**

`-v` affiche des informations sur la mémoire virtuelle. **BSD**

`-x` informe sur les processus sans terminaux. **BSD**

18.2 jobs

Liste les processus d'arrière-plan.

```
jobs [-l]
```

Affiche une ligne par processus, donnant son numéro de job et son statut.

Options :

`-l` liste aussi les numéros de processus.

18.3 at batch

Gestion de jobs batch et de processus différés.

```
at [-csm][-f fichier][time [date][+inc]                    SVR4
at [-csm] time [date][+inc][script]                        BSD
at {-r jobs...|-l [jobs...]}
batch [-csm][script]
```

Options :

`-c` c-shell exécute le script.

`-f` prend le script dans le *fichier*. **SVR4**

`-s` bourne shell exécute le script.

`-m` envoie un mail à la fin, même en cas de succès.

`-r` enlève les jobs spécifiés des queues *at* et *batch*.

`-l` liste les jobs spécifiés de l'utilisateur.

`at` permet l'exécution différée et contrôlée de processus.

`batch` place un processus dans une queue, et l'exécute lorsque la charge du système le permet.

Les processus à exécuter doivent être placés dans un script shell. Si le script n'est pas précisé, ils sont pris sur l'entrée standard.

La date et l'heure peuvent être spécifiées de différentes manières. Se reporter à la page de manuel.

```
pegase:~ 25) at 2015 next week my_dump_script
pegase:~ 26) at 8 :15am Jan 19 happy_birthday_script
pegase:~ 27) at now + 1 day
at> echo "1 elapsed day"
at> <EOT>
job 2736 at Sun Dec  2 15 :50 :00 1990
```

Chapitre 19
Grid

Grid est une couche de programmes permettant de distribuer des travaux sur une grille d'ordinateur (réseau de machine ou en anglais "grid"). Cette attribution de travaux sur des ordinateurs permet -si celle-ci est bien realisée- de charger les differents ordinateurs de manière équitable selon la politique choisie.

Le système Grid est basé sur l'architecture UNIX, il vous attribue donc des droits et peut ainsi vous refuser des commandes si vous n'avez pas les privilège requis

Grid est constitué de plusieurs fonctions d'un point de vue utilisateur :

19.1 fonction interface graphique général

`qmon` Interface graphique pilotant toutes les autres fonctions.

19.2 fonctions de contrôle

`qstat` affiche l'état des travaux des utilisateurs.

Options :

`[-u user_list]` ne montre que les entrées appartenant aux utilisateurs de la liste

`[-s p|r|s|z]` n'affiche que les états en attente (Pending), en cours de route (Running), en suspens (Suspended), orphelin (Zombie)

`qdel` supprime le travail (comme un `<CTRL-C>` dans un shell)

Options :

`[-f]` force la suppression

`[-verify]` simule la suppression

`[all]` supprime tout les travaux associés à un utilisateur

`qhold` gèle le travail (comme un `<CTRL-Z>` dans un shell)

`qrls` dégèle un travail

19.3 Soumission de batchs

Batch : Tâche travaillant en fond sans besoin d'interaction humaine (comme un "bg" dans un shell)

`qsub` soumet une tâche sur Grid

`qalter` change les options d'un travail déjà soumis

`qresub` crée une nouvelle tâche en recopiant une de celles qui sont en train de tourner ou en suspens

19.4 Soumission de job interactifs

Job interactif : Tâche ayant besoin d'interaction avec l'utilisateur (comme un "fg" sur un shell)

`qlogin` démarre une session du type telnet avec une selection automatique d'un ordinateur ayant les spécificités requises

`qsh` ouvre une session dans une nouvelle fenêtre

Remarque : si l'option -V est mise,il faut s'assurer que la variable d'environnement `DISPLAY` soit de la forme `{$hostname}:0.0` (ex :`obssb39:0.0`) et qu'on ait autorisé une machine distante à ouvrir une fenêtre X. Il faut utiliser xhost à cette intention (taper :`xhost +`)

`qtcsh` ouvre une session tcsh sur un ordinateur ayant les spécificités requises

19.5 Options des fonctions de soumission

`[-N nom_travail]` nom donné au travail dans Grid

`[-v variable_environnement]` passe la variable d'environnement sur le nouveau shell invoqué.

`[-V]` copie toutes les variables d'environnement sur le nouveau shell

`[-cwd]` fixe le répertoire courant de travail (cwd :Current Working Directory)

`[-l resource]` spécifie les resources dont le travail a besoin (mémoire, license d'outils, architecture...)

`[-S chemin_shell]` indique le chemin où se trouve le shell à lancer

19.6 Options relatives aux mails

`[-M dest]` indique les destinataire du mail.

`[-m b|e|a|s|n]` spécifie à quelle occasion un mail sera envoyé

b : au début(Begin) du travail

e : à la fin(End) du travail

a : lorsque le travail s'interompt(Abort)

s : lorsque le travail a été mise en suspens(Suspend)

n : en aucun cas(No)

N.B. Il est possible de mettre les options de soumissions dans le batch lui-même. Pour cela, il suffit de commencer la ligne du script avec #$. Ce préfixe indique donc à Grid que le texte le suivant lui est destiné.

19.7 Exemple

```
obssb39:~briner 140> ugvt grid
obssb39:~briner 140> xhost+
obssb39:~briner 140> setenv DISPLAY obssb39:0.0 #remarque sous qsh
obssb39:~briner 140> qsh -V -l rdb=TRUE -l mem_free=100M -l num_proc=2
 -M root@tree.ca -m b -S /bin/csh -cwd - N monTravail
```

Dans ce cas, on indique à Grid qu'on désire une fenêtre(qsh) sur une machine ayant :
- une license rdb(-l rdb=TRUE)
- 100Mo de mémoire libre(mem_free=100M)
- 2 processeurs

qu'un mail doit être envoyé à root@tree.ca(-M root@...) au lancement du travail(-m b)
que le shell doit être le cshell sous /bin(-S /bin/csh) et
que le le travail s'appellera pour Grid "monTravail".
De plus notez bien les différents points à effectuer avant ce genre de commande :
- rentrer dans l'environnement grid à l'aide de la commande "ugvt grid"
- spécifier au système qu'on autorise un client X distant à ouvrir une fenêtre sur notre bureau à l'aide de "xhost +"
- mettre à jour la variable d'environnement `DISPLAY` à `${hostname}:0.0` (`obssb39:0.0`) de manière à ce que la machine accueillant le travail redirige son client X sur la machine emmettante.

qu'on force la variable d'environnement `DISPLAY` à `obssb39:0.0` qui dans ce cas affiche la fenêtre sur notre propre machine .

19.8 Avantages offerts par Grid

- Distribue de manière quasi transparente les travaux tout en cachant les mécanismes de décision et en laissant à l'utilisateur la spécification des besoins pour la réalisation du travail(mémoire libre, taille du disque dur, l'architecture, license d'outils...)

- Met à disposition une série d'outils permettant de s'affranchir d'un arrêt impromptu d'une machine ou d'un manque de ressource temporel de celle-ci.

- Simplifie la gestion d'une série de travaux effectuant exactement le même calcul sur des données différentes les une des autres.

- Expose un interface de contrôle à l'utilisateur à travers la commande qmon.

Exercices du chapitre 19 :

1. Combien avez-vous de processus actifs en ce moment ?
2. Comment savoir si toutes les taches de fond sont terminées ?

Chapitre 20
Utilisation du réseau

Le réseau de l'Université et le réseau internet mondial sont accessibles à partir des stations de travail, pour se connecter sur d'autres machines ou transférer des fichiers ou d'autres informations.

Ces dernières années, malheureusement, les problèmes de sécurité (hackers etc.) compliquent beaucoup ces accès à distance. Pour cette raison, nous vous recommandons de ne jamais utiliser les connections non sécurisées, surtout dans un environnement autre que l'Observatoire. Il n'est pas difficile d'«écouter» sur le réseau et de capturer les mots de passe qui le traversent...

20.1 Connexions sécurisées

Les commandes `rlogin`, `rsh`, `rcp` et `ftp` ne sont pas sécurisées. De plus en plus de machines les refusent. Il vaut mieux utiliser leurs versions correspondantes sécurisées `slogin`, `ssh`, `scp` et `sftp`. Ces commandes commençant par un `s` cryptent toutes les informations sensibles. De plus elles peuvent compresser les messages, d'où un gain de vitesse sur les lignes lentes.

Une autre option consiste à utiliser des mots de passe à usage unique. Ce système est aussi disponible à l'Observatoire. Il permet d'accéder à nos stations à partir de n'importe quel point dans le monde, cyber café dans la brousse compris, sans risquer de compromettre notre sécurité.

20.2 `slogin`

Pour se connecter sur une autre machine UNIX.

`slogin [-l` *utilisateur*`]` *cible*

La machine *cible* doit être connue par les NIS(+) ou être donnée comme une adresse internet complète (par exemple : `phoebus.unige.ch`). Si l'on n'est pas listé dans le fichier `.rhosts` ou

`.shosts` de l'*utilisateur*, on devra entrer le mot de passe. Par défaut, l'*utilisateur* est le même que sur la machine locale.

Options :

`-l` permet de se connecter sous un autre nom d'*utilisateur*,

`-C` utilise la compression,

`-c blowfish` utilise **blowfish** pour crypter les messages (plus rapide et peut-être plus fiable que le système par défaut).

Exemples :

```
indiana:~ 12) slogin pegase.unige.ch
```

20.3 `ssh`

Pour exécuter une commande sur une autre machine UNIX.

`ssh [-l` *utilisateur*`] [-n]` *cible commande*

La machine *cible* doit être connue par les NIS(+) ou être donnée comme une adresse internet complète. On doit être listé dans le fichier `.shosts` de l'*utilisateur*. Par défaut, l'*utilisateur* est le même que sur la machine locale.

Options :

`-l` permet d'utiliser un autre nom d'*utilisateur*.

`-n` permet de rediriger l'entrée du `ssh` sur `/dev/null`, et d'éviter ainsi certaines interactions entre le shell et le `ssh`, qui pourraient bloquer la commande.

`-C` utilise la compression,

`-c blowfish` utilise **blowfish** pour crypter les messages (plus rapide et peut-être plus fiable que le système par défaut).

20.4 `telnet`

Pour se connecter sur une autre machine UNIX ou non, supportant le protocole TCP/IP. N'est pas sécurisée.

`telnet` *cible*

La machine *cible* doit être connue par les NIS(+) ou être donnée comme une adresse internet complète. On devra entrer le mot de passe.

On peut lancer `telnet` depuis n'importe quel `shell`, mais il est parfois plus commode, dans les environnements de fenêtres, d'ouvrir une fenêtre émulant un terminal connu et fonctionnant bien sur la machine *cible*.

En dehors de l'Observatoire, il est impératif d'utiliser les mots de passe à usage unique (OTP)

Sous X11 existe un autre émulateur de terminal, qu'on lance avec la commande

`xterm`

Cet émulateur possède une fenêtre VT100, et une fenêtre Tektronix 4010-4014. **On peut dans la fenêtre activer des menus relatifs à l'émulateur de terminal `xterm` en appuyant simultanément sur la touche `<CTRL>`, et sur un bouton de la souris**.

Le bouton de droite donne accès aux menu de fontes. Le bouton du milieu permet des réglages sur la configuration du terminal émulé, ainsi que la commutation VT — TEK. Le bouton de gauche ouvre le menu principal de gestion du programme.

20.5 scp

Pour transférer des fichiers avec une autre machine UNIX.

`scp [-p]` *fichier1 fichier2*
`scp [-pr]` *fichier...répertoire*

Le *fichier1* est copié dans le *fichier2*, ou les *fichiers* sont copiés dans le *répertoire*.

Options :

`-p` permet de copier le fichier et ses attributs.

`-r` option de récursivité, la destination doit être un répertoire.

Le fichier (ou répertoire) local est spécifié normalement, alors que le nom du fichier (ou répertoire) situé sur l'autre machine doit être précédé du nom de la machine *cible*, et éventuellement d'un nom d'utilisateur.

[*utilisateur*`@`] *cible* `:`*fichier*

La *machine cible* doit être connue par les NIS(+) ou être donnée comme une adresse internet complète. On doit être listé dans le fichier `.shosts` de l'*utilisateur*. Par défaut, l'*utilisateur* est le même que sur la machine locale.

Exemples :

```
hercule:~ 4) scp pegase:/.tcshrc
```

20.6 sftp

Pour transférer des fichiers avec une machine UNIX ou non, supportant le protocole TCP/IP.

`sftp` *machine*

La *machine cible* doit être connue par les NIS(+) ou être donnée comme une adresse internet complète. On devra entrer le mot de passe.

On peut ensuite donner des commandes pour transférer des fichiers, lister des répertoires, ou régler des paramètres dans un mode interactif signalé par le prompt

sftp>

Les commandes principales sont :

`help`	aide par la machine *locale*.
`remotehelp`	aide par la machine *cible*.
`put` *file*	envoi d'un fichier vers l'autre machine.
`mput` *file...*	envoi de plusieurs fichiers vers l'autre machine.
`get` *file*	envoi d'un fichier de l'autre machine.
`mget` *file...*	envoi de plusieurs fichiers de l'autre machine.
`prompt`	commutation du mode interactif (confirmations).
`dir`	liste le contenu du répertoire.
`lcd` *newdir*	change de répertoire *local*.
`cd` *newdir*	change de répertoire dans la machine *cible*.

`binary`	initialise le type binaire pour le transfert de fichiers.
`ascii`	initialise le type ascii pour le transfert de fichiers.
`!`*commande*	sort dans le shell et exécute la *commande*.

Chapitre 21
Le Shell

Le Shell est un programme de dialogue entre l'utilisateur et le système. Le paragraphe suivant décrit les tâches principales du shell. L'utilisateur a le choix entre plusieurs shells, qui sont évoqués plus loin.

21.1 Les tâches du Shell

– l'exécution des programmes

```
pegase:~ 29) lprtxt fichier  ; lpq +1
```

– les substitutions

```
pegase:~ 30) cd $home
pegase:~ 31) ls *.c
```

– la redirection des entrées–sorties

```
pegase:~ 32) ls -alg > liste
```

– les « tubes » (connection des entrées–sorties)

```
pegase:~ 33) ls | wc
```

– les variables d'environnement

```
pegase:~ 34) setenv HOME /home/system/tartempion
```

– l'interprétation des commandes

```
pegase:~ 35) foreach file ( *.c )
> echo $file
> end
pegase:~ 36) if ( -e repertoire ) cd repertoire
```

21.2 Le Bourne Shell

`sh` (*Bourne Shell*) est l'interpréteur de commande standard d'UNIX. Il est plus rapide que d'autres shell à l'interprétation des commandes, mais possède de moins grandes facilités, notamment en utilisation interactive. Il est principalement utilisé dans des scripts shell (fichiers de commandes) pour lesquels il est efficace, notamment grâce à ses possibilités de programmation (fonctions, etc.).

Il offre des facilités comme :

– *substitutions* de variables

```
$ string="Bonjour"
$ echo $string
```

– *substitutions* de commandes

```
$ echo "Le contenu est  :‘ls‘"
```

– *substitutions* de noms de fichiers

```
$ ls *.tex
```

21.3 Le C-Shell

`csh` (*C-Shell*) est un interpréteur de commandes, dont la syntaxe est proche du C.

Il offre les facilités du *Bourne Shell* plus un certain nombre de facilités nouvelles :

– *achèvement* des noms de fichiers si la variable `filec` est définie.

```
pegase% set filec
pegase% cd /ho<ESC>
```

donne

```
pegase% cd /home
```

– *achèvement* des noms d'utilisateurs

```
pegase% cd ~tart<ESC>
```

donne

```
pegase% cd ~tartempion
```

– définition d'*alias*

```
pegase% alias dir /bin/ls -al
```

– *historique* des commandes

```
pegase% history
   1  set filec
   2  cd /home
   3  cd ~tartempion
   4  alias dir /bin/ls -al
   5  history
```

– *substitutions* historiques

```
pegase% dir !3 :$
dir ~tartempion
...
```

– historique sauvé entre les sessions dans le fichier `~/.history`

```
pegase% set savehist=20
```

21.4 Le TC-Shell

`tcsh` est un C-shell, et toute la description du C-Shell faite plus loin est valable pour le TC-Shell. Seul le comportement interactif du TC-Shell a été amélioré par l'addition de certaines facilités, dont :

– *édition* de la ligne de commandes (voir Emacs)

```
pegase:~ 37) cd /home/systeme/tartempion
/home/systeme/tartempion : No such file or directory
pegase:~ 38) cd /home/systeme/tartempion<M-B><C-B><DEL>
```

donne

```
pegase:~ 38) cd /home/system/tartempion
```

– *achèvement* des noms de commandes

```
pegase:~ 39) lprt<TAB>
```

donne

```
pegase:~ 39) lprtxt
```

– *correction orthographique* des noms de commandes, de fichiers ou d'utilisateurs.

```
pegase:~ 40) cd ~tartepion<M-$>
```

donne

```
pegase:~ 40) cd ~tartempion
```

– *déplacement visuel* dans l'historique

– etc ...

21.5 Démarrage et fin du C-Shell

A l'initialisation, le C-Shell exécute les commandes du fichier `.cshrc`, qui exécute lui-même `.alias`.

Si un shell est lancé par le processus `login`, il est appelé *login shell*. Dans ce cas, après avoir exécuté les commandes du fichier `.cshrc`, et du fichier `.alias`, il exécute les commandes du fichier `.login`.

A la fin d'un *login shell*, les commandes du fichier `.logout` sont exécutées.

21.5.1 Fichiers locaux

A l'Observatoire, les fichiers `.login` , `.logout` , `.cshrc` et `.alias` sont définis de manière identique pour chacun, et font appel à des fichiers locaux pour les initialisations personnelles, de manière à pouvoir être modifiés par les responsables du système.

Pour ne pas risquer de perdre vos modifications ou initialisations personnelles, **mettez les** dans les fichiers `.login.local` , `.logout.local` , `.cshrc.local` et `.alias.local` , qui sont appelés automatiquement s'ils existent, à la fin des fichiers communs.

21.5.2 C-Shell interactif

Après le démarrage, un shell interactif lit les commandes du terminal.

Pour cela, il exécute répétitivement :

– *Lecture* d'une ligne de commandes
– *Découpage* en mots
– *Enregistrement* dans l'historique
– *Analyse* de la séquence de mots
– *Exécution* de chaque commande

21.5.3 C-Shell non-interactif

Un shell non-interactif ne lit pas les commandes du terminal.

Il les reçoit comme paramètres sur la ligne de commande, ou va les chercher dans un script shell (fichier de commandes) qui doit être éxecutable.

`csh -c` *fichier* exécute les commandes d'un script shell dans un sous-processus.
`source` *fichier* permet de lire les commandes d'un script shell, sans créer de sous-processus.

21.6 Achèvement de noms

La variable *filec* permet, si elle est définie, au C-shell de tenter l'achèvement des noms de fichiers.

Un nom partiel et non-ambigu de fichier suivi d'un `<ESC>` sera alors résolu et complété à l'écran.

Si un nom partiel de fichier est suivi d'un `<CTRL>-D`, le C-shell donnera une liste de toutes les possibilités, avant de réafficher la ligne incomplète.

Si un mot incomplet commence par un `~`, le C-shell tentera de le compléter par un nom d'utilisateur et non de fichier.

21.7 Structure lexicale

Le shell découpe la ligne en mots en utilisant comme séparateurs l'espace et le tabulateur.

Les caractères `&`, `|`, `;`, `<`, `>`, `(`, et `)` sont appelés métacaractères et forment des mots séparés

Les paires de ces caractères forment aussi des mots : `||`, `&&`, `|&`, `>&` ...

Si l'on désire utiliser un de ces caractères à l'intérieur d'un mot, il suffit de mettre un backslash (\) devant.

Les chaînes de caractères entourées de single quote (’), double quote (") ou back quote (‘) forment des mots partiels. Dans de telles chaînes, les métacaractères ne forment pas des mots separés.

Dans un script shell, le caractère # introduit un commentaire jusqu'à la fin de la ligne.

Un `<NEWLINE>` précédé par un \ est équivalent à un `<SPACE>`.

Remarque : A l'intérieur de back quotes (‘) ou de double quotes ("), un `<NEWLINE>` précédé par un \ donne un vrai `<NEWLINE>`.

21.8 Analyse de la ligne

Une commande simple est composée d'une séquence de mots.
– Le premier mot spécifie la commande à exécuter.
– Le mot `;` permet de regrouper en une seule plusieurs commandes simples, qui seront exécutées séquentiellement.

– Les mots `|` et `|&` forment un tube (pipe). Avec `|`, la sortie standard de la commande précédente est redirigée dans l'entrée standard de la commande qui suit. Avec `|&`, la sortie des erreurs *et* la sortie standard sont redirigées dans le tube.
– Les mots `&&` et `||` forment un tube conditionnel. L'exécution de la partie droite du tube dépend de la réussite ou de l'échec de la partie gauche du tube.
– Un tube ou une séquence de commandes peut être entourée de parenthèses `(...)` pour former une simple commande qui peut être un composant dans un tube ou dans une séquence.
– Une commande peut être exécutée de manière asynchrone (en background) en ajoutant à la fin de la commande le mot `&`.

21.8.1 Le single quote

Aucun des caractères spéciaux du shell n'est interprété à l'intérieur des apostrophes (single quotes), à l'exception du « `!` », et du « \ » qui permet de le protéger :

```
pegase:~ 41) echo '1          2'
1          2
pegase:~ 42) echo '< > |  ; ( ) { \ }  >> <<  " ` & \!'
< > |  ; ( ) { \ }  >> <<  " ` & !
```

21.8.2 Le double quote

Les seuls caractères spéciaux qui sont interprétés à l'intérieur de guillemets (double quotes) sont :
– `$` le signe dollar (substitution de variable).
– `` ` `` le back quote (substitution de commande).
– `!` le point d'exclamation (substitution historique).
– \ le backslash (caractère d'échappement).

21.8.3 Le backslash

Le caractère d'échappement \ sert à ôter la signification particulière des caractères spéciaux.

```
pegase:~ 43) echo \>
>
pegase:~ 44) echo \\
\
pegase:~ 45) set x=5
pegase:~ 46) echo \$x=$x
$x=5
pegase:~ 47) echo x = \"$x\"  ; echo 'x = "$x"'
x = "5" x = "$x"
```

21.9 Substitution des commandes - le back quote

Une commande entourée par des back quotes (`` `...` ``) est exécutée par un sous-shell, dont la sortie standard sert de chaîne de remplacement.

Un groupe de `<SPACE>`, `<TAB>` et de `<NEWLINE>` est remplacé par un `<SPACE>` unique. Si des double quotes entourent les back quotes, seul le caractère `<NEWLINE>` est remplacé par un `<SPACE>`.

```
pegase:~ 48) echo le directory contient `ls`
```

21.10 Substitution historique

Les mots commencant par `!` font référence à une substitution historique, excepté si le caractère suivant est le `<SPACE>`, `<TAB>`, `<NEWLINE>`, `=` ou `)`

Désignateurs d'événements :
- `!!` référencie la dernière commande
- `!n` référencie la n$^{\text{ième}}$ commande
- `!-n` référencie la commande courante moins n
- `!str` référencie la dernière commande commencant par `str`
- `!?str` référencie la dernière commande contenant `str`

La référence à substituer peut n'être qu'un mot d'une commande ; les désignateurs de mots suivants (facultatifs) doivent être séparés par `" :"`
- `0` référencie le premier mot (commande)
- `n` référencie le n$^{\text{ième}}$ argument
- `^` référencie le premier argument
- `$` référencie le dernier mot
- `*` référencie tous les arguments

On peut enfin modifier la référence ainsi formée, les modificateurs suivants (facultatifs et cumulables) sont aussi séparés par des `" :"`
- `h` enlève le dernier composant du pathname
- `t` ne garde que ce dernier composant.
- `r` enlève l'extension du nom de fichier
- `e` ne garde que cette extension
- `s/`*`s1`*`/`*`s2`*`/` substitue la première occurence de '*s1*' par '*s2*'
- `gs/`*`s1`*`/`*`s2`*`/` substitue toutes les occurences de '*s1*' par '*s2*'

Exemple :

```
pegase:~ 49) ls /usr/local/bin/b*
/usr/local/bin/bed*       /usr/local/bin/brushtopbm*
/usr/local/bin/bibtex*    /usr/local/bin/btoa*
pegase:~ 50) !$ :h/bibtex myfile
/usr/local/bin/bibtex myfile
```

21.11 `alias`

Le C-shell maintient une liste d'alias que l'on peut créer, afficher et modifier en utilisant les commandes `alias` et `unalias`. `alias` sans paramètre liste les alias définis.

Exemples :

```
pegase:~ 51) alias ls /bin/ls -F
pegase:~ 52) alias lo exit
pegase:~ 53) alias lo
exit
pegase:~ 54) unalias lo
pegase:~ 55) alias lo
pegase:~ 56) alias h history
```

Le shell examine le premier mot de la ligne de commande pour voir s'il correspond à un nom d'alias existant. Si c'est le cas, il substitue l'alias et réinterprète la ligne de commande.

Les alias peuvent être imbriqués dans d'autres alias.

21.12 Redirection des entrées–sorties

– `<` redirige un fichier sur l'entrée standard.

```
pegase:~ 57) cat < /var/adm/messages
...
```

– `<< word` lit l'entrée standard, et place le résultat dans un fichier temporaire, jusqu'à ce qu'une ligne soit identique à `word`.

```
pegase:~ 58) cat <<  ********
******************
‘date‘
******************
********
******************
Sun Dec  2 19 :00 :30 MET 1990
******************
```

– `>` redirige la sortie standard dans un fichier.

```
pegase:~ 59) ls > /tmp/contenu
```

– `>&` redirige la sortie standard *et* la sortie des erreurs dans un fichier.

– `>>` ajoute la sortie standard à la fin d'un fichier.

```
pegase:~ 60) echo 'contenu :' > /tmp/contenu
pegase:~ 61) ls  >> /tmp/contenu
```

– `>>&` ajoute la sortie standard *et* la sortie des erreurs à la fin d'un fichier.

Pour rediriger seulement la sortie des erreurs :

```
( command > outfile ) >& errorfile
```

21.12.1 `tee`

On peut à la fois rediriger la sortie standard sur des fichiers et la laisser sortir normalement, avec la commande :

`tee [ -a ] [` *fichier* `] ...`

`tee` transfère l'entrée standard sur la sortie standard, et en fait eventuellement une copie dans les fichiers spécifiés. L'option `-a` permet d'ajouter à la fin des fichiers au lieu de les remplacer.

On utilise donc cette commande dans un tube :

```
pegase:~ 62) tee entree <<  *** | wc  ; cat entree
ligne 1
ligne 2
***
    2    4    16
ligne 1
ligne 2
pegase:~ 63) ls -1 | tee contenu | echo "j'ai ‘wc -l‘ fichiers \!"
j'ai    46 fichiers !
```

21.13 Variables

Une variable est composée d'un nom et d'une valeur. Il existe deux types de variables :

– les variables locales, créées, affichées et détruites par les commandes `set` et `unset`, qui ne sont connues que du shell dans lequel elles sont définies :

```
pegase:~ 64) set string='ce n\'est qu\'un "mot"'
pegase:~ 65) set chaine="ce n'est qu'un \"mot\""
pegase:~ 66) set phrase=(une jolie petite phrase)
```

– les variables d'environnement (transmises aux processus *fils*), gerées par les commandes `setenv`, `printenv` et `unsetenv` :

```
pegase:~ 67) setenv MY_ENV_VAR "variable globale"
pegase:~ 68) printenv MY_ENV_VAR
variable globale
```

21.13.1 Substitution des variables

Pour substituer sa valeur, il suffit de taper le nom d'une variable précédé d'un `$`.

La substitution peut être supprimée en mettant un \ devant le `$`, sauf à l'intérieur de double quotes.

```
pegase:~ 69) echo $chaine
ce n'est qu'un "mot"
pegase:~ 70) echo $phrase[1-3]
une jolie petite
pegase:~ 71) echo $#phrase                                  (nombre d'éléments)
4
pegase:~ 72) echo $?phrase                                  (test d'existence)
1
pegase:~ 73) echo -n "entrez une donnee : " ;\
echo la reponse est $<
entrez une donnee : 31
la reponse est 31
```

21.13.2 Variables relatives au shell

Un certain nombre de variables sont maintenues et utilisées par le shell (interactif ou non).

– `$0` substitue le nom du programme
– `$n` est équivalent à `$argv[n]`, c'est-à-dire le composant `n` de la ligne de commande appelant le shell. `$0` est le nom de la commande, `$1` et suivants sont les paramètres.
– `$*` est équivalent à `$argv[*]`, c'est-à-dire une chaîne contenant tous les paramètres de la ligne de commande.
– `$$` substitue le numéro du processus (pid)
– `$<` substitue une ligne de l'entrée standard
– `argv` : la liste des arguments du shell.
– `cdpath` : la liste des répertoires où chercher un sous-répertoire avec `cd`, `chdir` ou `popd`.
– `cwd` : le répertoire de travail courant.
– `history` : le nombre de commandes à conserver dans ce shell.
– `savehist` : le nombre de commandes à conserver entre shells.

– `home` : le répertoire de base de l'utilisateur.
– `path` : la liste des répertoires où chercher les exécutables.
– `status` le status retourné par la dernière commande.

21.14 Substitution des noms de fichiers

Les mots non-protégés (contenus entre des apostrophes ou des guillemets), contenant `*`, `?`, `[`, `{` ou commençant par `~` sont developpés en une liste de noms de fichiers répondants aux conditions imposées par les métacaractères :
– `*` zéro ou plus de caractères.
– `?` un seul caractère.
– `[...]` un des caractères se trouvant entre les crochets.
– `{ str, str, ... }` une des chaînes se trouvant dans les accolades.
– `~` le chemin du répertoire de base (*home directory*) de l'utilisateur.
– `~drofnats` substitue le répertoire de base de l'utilisateur `drofnats`.
Les caractères `.` et `/` ne peuvent pas être remplacés par des métacaractères.

La variable *noglob*, si elle est définie, empêche la substitution des noms de fichiers.

21.15 Expressions et opérateurs

Un certain nombre de commandes du shell utilisent des expressions, dont le résultat donne la valeur ou le status de la commande.

Opérateurs purement numériques :
– arithmétiques : `+`, `-`, `*`, `/`, `%` (modulo), `~` (complément à 1)
– de comparaison : `<`, `>`, `<=`, `>=`
– de manipulation de bits : `&` (et), `|` (ou), `^` (ou exclusif), `<<` , `>>` (décalages)
– logiques : `&&` (et), `||` (ou), `!` (négation)
Opérateurs numériques et de chaînes de caractères :
– de relation : `==` (égalité) `!=` (inégalité) `=~` (correspondance) `!~` (non-correspondance)

```
# ! /bin/csh -f
foreach i (*)
if ($i =~ *.tex) echo $i : fichier TeX
end
```

Opérateurs sur les fichiers :
`-r` *vrai* si l'utilisateur peut le lire.
`-w` *vrai* si l'utilisateur peut y écrire.
`-x` *vrai* si l'utilisateur peut l'exécuter.
`-e` *vrai* si le fichier existe.
`-o` *vrai* si l'utilisateur en est le propriétaire.
`-z` *vrai* si le fichier est vide.
`-f` *vrai* si c'est un fichier ordinaire.
`-d` *vrai* si le fichier est un répertoire.

```
if (-e ~/.login.local) source ~/.login.local
if (-d /tftpboot) then
    rarpd -a  ; rpc  bootparamd  ; echo -n ' rarpd'
fi
```

21.15.1 `@`

Il est possible d'effectuer des calculs en utilisant :
- les variables
- les opérateurs arithmétiques
- les opérateurs de manipulation de bits
- les opérateurs `++` (incrémentation) et `--` (décrementation)

`@` *variable* `=` *expression*
`@` *variable* `++|--`

Le raccourci :

`@` *variable op*`=` *expression*

peut être utilisé avec les opérateurs `+`, `-`, `*`, `/`, `%`, ou `^` pour

`@` *variable* `=` *variable op* `(` *expression* `)`

Exemples :

```
pegase:~ 74) @ groupe = ( $uid / 100 ) * 100
pegase:~ 75) @ numero = $uid - $groupe
pegase:~ 76) echo $numero
15 pegase:~ 77) @ numero --
pegase:~ 78) echo $numero
14
```

21.16 Les branchements

Un certain nombre de commandes de branchements permettent de modifier le déroulement des commandes du shell.

Ces commandes sont utilisées principalement dans des shells non-interactifs (script shell).

Dans toutes les commandes de branchements, les mots-clés doivent être seuls sur une ligne, sauf mentionné expressément.

21.16.1 `if`

La commande d'exécution conditionnelle. Elle prend deux formes :

La forme simple : *command* doit être entièrement sur la même ligne que `if` :

`if (`*expression*`)` *command*

Sa forme plus élaborée, qui permet d'exécuter conditionnellement plusieurs commandes :

```
if (expression) then
   ...
[else if (expression) then
   ...]
[else
   ...]
endif
```

21.16.2 switch

Chaque *pattern* est comparé à *string*. Lorsque la correspondance est établie, les commandes sont exécutées. Le `breaksw` permet de finir le `switch`. Les `case` *pattern :* et le label `default :` doivent être placés au début d'une ligne.

```
switch (string)
case pattern1 :
   ...
   breaksw
case pattern2 :
   ...
   breaksw
...
[default :
   ...]
endsw
```

21.16.3 foreach

La boucle est pilotée par une variable. Chaque valeur de *list* est attribuée successivement à *var*, et la boucle est exécutée :

```
foreach var (list)
   ...
end
```

Exemple :

```
pegase:~ 79) foreach fichier (`ls -1 | grep .tex`)
? latex $fichier
? end
```

Si *list* est vide, et qu'il peut le determiner, `foreach` le signale immédiatement.

21.16.4 while

La boucle conditionnelle. Tant que *expression* est vraie (résultat non-nul), la boucle est parcourue, et ses commandes exécutées.

```
while (expression)
   ...
end
```

Exemple :

```
pegase:~ 80) @ i = 4701
pegase:~ 81) while ( $i < 4800 )
? niscat passwd | grep $i |\
   awk -F : '{printf("%d : %s est %s\n",$3,$5,$1)}'
? @ i++
? end
...
```

21.16.5 Autres commandes de branchements

– Pour continuer l'exécution sur la ligne commençant par *label :* :

`goto` *label*

Exemple :

```
if ( status != 0 ) then     goto end endif ... ... end:     exit
```

– Pour sortir d'une boucle *while* ou *foreach*

`break`

Exemple :

```
pegase:~ 82) @ i = 4701
pegase:~ 83) while ( $i < 4800 )
? set un = ‘niscat passwd | grep $i‘
? if ( "$un" == "" ) break
? echo $un |\
   awk -F : ’{printf("%d : %s est %s\n",$3,$5,$1)}’
? @ i++
? end
...
```

– Pour continuer l'exécution au début de la boucle *while* ou *foreach*.

`continue`

21.17 Exécution des commandes

Les commandes internes du shell sont exécutées directement.

Les autres commandes sont des programmes. Le shell cherche un fichier dont le nom correspond.

Si le nom commence par un `/`, le shell le considère comme un chemin absolu de recherche.

Sinon, il le cherche dans les répertoires spécifiés par la variable *path*.

Le shell utilise une *table de recherche* (hash-table) pour éliminer les répertoires ne contenant pas de fichiers appropriés. On peut demander au shell de recalculer sa table de recherche, ou de ne pas en tenir compte avec les commandes :

```
rehash
```

et

```
unhash
```

Lorsque le shell a résolu l'accès au fichier, et vérifié les permissions d'exécution, il *crée* (fork) un *sous-processus* (child).

Le *noyau* (kernel) tente alors de remplacer ce nouveau shell par le programme appelé.

Si c'est un exécutable, pas de problèmes.

Si c'est un fichier texte, dont les deux premiers caractères sont `#!`, il utilise la fin de la première ligne comme ligne de commande pour l'exécution de ce script shell.

Sinon, il invoque le C-shell ou le Bourne-shell, selon que le premier caractère est ou non un `#`

21.17.1 Les priorités et `nice`

Chaque processus est exécuté avec une certaine priorité, représentée par un nombre entre 0 et 127. Plus la valeur est grande, moins haute est la priorité. A l'origine, chaque processus lancé

dans un C-Shell se voit attribuer une priorité de base.

La priorité de base d'un programme est modifiée au cours du temps par le système, sur la base du temps CPU utilisé, du temps d'attente, et d'autres paramètres

On peut modifier, par incrément n (entre -20 et +20), la priorité de base d'un processus avec la commande :

`nice [+`*n*`|-`*n*`]` *command*

- `+`*n* incrémente la valeur de priorité de *n*.
- `-`*n* décrémente la valeur de priorité de *n*.
- Si cet argument est omis, la valeur est 4.

Seul le super-user peut utiliser `-`***n***.

21.17.2 Quelques commandes liées à l'exécution

- Pour réévaluer *command*. Cette commande est utile lorsque l'on veut exécuter une commande qui résulte d'une substitution de variable ou de commande, puisque l'évaluation de la ligne a lieu avant les substitutions.

  `eval` *command*

  Exemple :

  ```
  pegase:~ 84) cat .login.local
  setenv TEXEDIT 'emacs +%d %s'
  echo ".login.local execute"
  pegase:~ 85) grep .login.local .login
  if (-e ~/.login.local) source ~/.login.local
  pegase:~ 86) `grep .login.local .login`
  `grep .login.local .login` : Ambiguous.
  pegase:~ 87) eval `grep .login.local .login`
  .login.local execute
  ```

- Pour répéter *count* fois *command* :

  `repeat` *count command*

- Pour décaler des mots se trouvant dans *argv* ou *variable*.

  `shift [`*variable*`]`

- Pour exécuter un fichier de commandes (script shell).

  `source` *fichier*

- Pour sortir d'un *script shell*.

  `exit [`*status*`]`

21.18 Les signaux

Le système UNIX fournit un certain nombre de signaux d'*interruption* pour communiquer entre les processus.

Un processus peut définir une routine de gestion des signaux, qui peut traiter, bloquer ou ignorer certains signaux.

On peut envoyer du clavier les signaux `SIGTERM (^C)` et `SIGTSTP (^Z)`, pour tuer ou stopper un processus interactif.

21.18.1 kill

Pour envoyer un signal *sig* intentionnellement à un processus. Si *sig* n'est pas spécifié (par son nom ou son numéro), le signal `SIGTERM` (15) est envoyé.

```
kill [-sig] {pid|%job}
kill [-l]
```

Le shell peut spécifier un masque bloquant certains signaux, qui ne sont dès lors plus envoyés aux processus. Un processus receveur peut traiter les autres signaux, ou les ignorer. Un signal qui n'est pas traité, bloqué ou ignoré a pour effet de tuer le processus.

Options :

`-l` liste les signaux.

Le signal `SIGKILL` (9) ne peut être traité, bloqué, ni ignoré.
Donc, `kill -9` *pid* est un bon moyen pour tuer un processus.

21.18.2 onintr

Il est possible dans un script shell d'exécuter des commandes lorsqu'un signal d'interruption arrive avec la commande `onintr`, ou d'ignorer tous les signaux d'interruption (option `-`).

```
onintr [ - | label ]
```

Exemple :
Le script shell suivant permet de contrôler la commande de traitement des interruptions :

```
# ! /bin/csh -f
set i=1
onintr erreur

while $i
echo Tapez un \<Control-C\> pour sortir\!\!
set val=$<  ; echo val : $val
end
exit

erreur :
echo Ca marche \!\!
```

21.19 Contrôle des jobs

Le shell associe un numéro de *job* à chaque processus tournant en background.

Lorsqu'on lance un processus asynchrone, le shell affiche une ligne :

```
[1] 389
```

Le nombre entre crochets est le numéro de job, le nombre suivant est le numéro de processus (`pid`).

Les références aux jobs commencent par le signe `%`

`%`, `%%`, `%+` référencient le job courant.
`%-` référencie le job précédent.
`%j` référencie le job identifié par `j`. `j` peut être le numéro du job ou une chaîne l'identifiant de manière unique.

21.19.1 Quelques commandes liées aux jobs

– Pour relancer un *job* en background.

```
bg [ job-reference ]
```

– Pour relancer un *job* spécifié en foreground.

```
fg [ job-reference ]
```

– Pour lister les *jobs* actifs.

```
jobs
```

– Pour attendre la fin d'un *processus* exécuté en background.

```
wait
```

21.20 Les différences du TC-shell

Le TC-Shell permet de rappeler les commandes de l'historique avec les touches du curseur.

On peut également éditer la ligne de commande, avec des commandes du type *emacs*.

L'achèvement des noms de fichiers, d'utilisateurs, et de commandes est commandé par la touche `TAB`, au lieu de `ESC`

On peut corriger l'orthographe des noms de fichiers, commandes ou d'utilisateurs avec la séquence `ESC $`

Chapitre 22

Les expressions régulières et la famille de `grep`

Un certain nombre d'outils UNIX permettent de travailler du texte en employant des caractères dont la signification est particulière, et qui, seuls ou en chaîne, constituent des *expressions régulières*.

Les outils UNIX remplacent ces expressions par des chaînes de caractères correspondantes avant de les traiter de manière appropriée.

22.1 Expressions simples

Les expression régulières simples sont employées par `grep`, `ed` et `sed`

Elles sont composées à partir des métacaractères suivants :

`^`	correspond au début d'une ligne, sauf s'il est situé directement après `[` .
`$`	correspond à la fin d'une ligne.
`.`	correspond à n'importe quel caractère, sauf *newline*.
`*`	correspond à 0 ou plus occurences du caractère précédent.
`[`*cars*`]`	correspond à n'importe quel caractère inclus entre les crochets.
`[^`*cars*`]`	correspond à n'importe quel caractère non-inclus entre les crochets.
`\c`	correspond au métacaractère `c`.

Exemples :

`^ *`	correspond à n'importe quel nombre de blancs en début de ligne.
`[0-9][a-z]*`	correspond à n'importe quel chiffre suivi de lettres minuscules ou de rien.
`[0-9][a-z ]`	correspond à n'importe quel chiffre suivi d'une lettre minuscule ou d'un blanc.
`[ABC][^!a-z]`	correspond à un A, B ou un C majuscule suivi de n'importe quoi, sauf d'une lettre minuscule ou d'un point d'exclamation.
`[0-9][0-9][0-9]\$$`	correspond à trois chiffres suivi du signe dollar en fin de ligne.

22.2 Expressions étendues

`awk`, `egrep`, et `emacs` utilisent d'autres expressions régulières.

En plus des expressions simples, les caractères spéciaux suivants sont à disposition pour créer des expressions régulières plus performantes.

`*`	correspond à 0 ou plus occurences du caractère ou motif précédent.
`+`	correspond à 1 ou plusieurs occurences du caractère ou motif précédent.
`?`	correspond à 0 ou 1 occurence du caractère précédent.
`|`	correspond au motif précédent ou au motif suivant.
`(...)`	permet de fabriquer des motifs complexes, par combinaison de caractères ou de motifs plus simples.

Exemples :
Utilisons la phrase « *Ah, Henri II apprend LaTeX 2.09 à son chien.* ». Les occurences trouvées sont mises en gras.

`chat|chien`	correspond à la chaîne `chat` ou à la chaîne `chien`. (*Ah, Henri II apprend LaTeX 2.09 à son* **chien.**)
`[0-9]+`	correspond à n'importe quel nombre entier d'au moins un chiffre. (*Ah, Henri II apprend LaTeX* **2.09** *à son chien.*)
`[A-Z][a-z]+`	correspond à n'importe quel suite d'une capitale suivie d'au moins une lettre minuscule. (**Ah**, **Henri** *II apprend* **LaTe***X 2.09 à son chien.*)
`([A-Z][a-z])+`	correspond à au moins une occurence d'une séquence lettre majuscule-lettre minuscule. (**Ah**, **He***nri II apprend* **LaTe***X 2.09 à son chien.*)

Remarquons que dans le troisième exemple, **LaTe** donne deux correspondances de mot capitalisé de deux lettres, alors que dans le quatrième exemple, il donne une correspondance de double occurence de la séquence.

22.3 `grep` `egrep` `fgrep`

Recherche de chaîne de caractères dans des fichiers

`grep [-chilnv]` *motif* `[`*fichier...*`]`
`egrep [-chilnv]` *motif* `[`*fichier...*`]`

`fgrep [-chilnvx]` *chaîne* [*fichier...*]

`grep` recherche dans chaque fichier les lignes contenant *motif*, et les imprime sur la sortie standard. Le *motif* est une expression régulière simple.

`egrep` est une version étendue et plus rapide de grep. Le *motif* est une expression régulière étendue. Le mécanisme de recherche est différent et peut être un ordre de grandeur plus rapide. On l'utilisera de préference aux autres, lorsque l'option `x` n'est pas nécessaire.

`fgrep` permet la recherche de chaînes fixes. Les expressions régulières ne sont pas supportées.

Options :

`-c` imprime le nombre de lignes correspondantes, à la place des lignes elles-même.

`-h` supprime l'impression du nom du fichier.

`-i` ignore les différences minuscules-majuscules.

`-l` imprime le nom des fichiers contenant les lignes correspondantes à la place de celles-ci.

`-n` imprime également le numéro des lignes trouvées.

`-v` imprime les lignes ne contenant pas le *motif*.

`-x` imprime les lignes exactement identiques à la chaîne. Uniquement pour `fgrep`.

Exemples :
Trouvons les processus faisant appel aux « remote procedure call » (processus démons NFS).

```
pegase:~ 88) ps -ax | grep rpc | grep -v grep
    99 ?  IW    0 :11 rpc.mountd -n
   106 ?  IW    0 :05 rpc.bootparamd
   110 ?  IW    0 :00 rpc.statd
   111 ?  IW    0 :00 rpc.lockd
   300 ?  IW    0 :11 rpc.rquotad
```

Affichons le nombre d'utilisateurs dont le prénom commence par un P majuscule.

La première commande donne le nombre de lignes contenant un P majuscule dans le fichier `/etc/passwd` des NIS. Il peut être dans le nom, le prénom ou dans le mot de passe encrypté. La commande suivante élimine le nom, qui ne contient que des majuscules, enfin, la dernière commande impose un blanc avant le P majuscule, ce qui élimine le champ de mot de passe encrypté.

```
pegase:~ 89) niscat passwd | egrep -c 'P[a-z]*'
83
pegase:~ 90) niscat passwd | egrep -c 'P[a-z]+'
32
pegase:~ 91) niscat passwd | egrep -c ' P[a-z]+'
12
```

Chapitre 23

awk

`awk` est un langage de traitement de motifs de caractères.

`awk [-F`*fscar*`] [-f` *script*`] [`*commandes*`] [`*fichier...*`]`

Un programme `awk` peut comprendre autant d'instructions que nécessaire. Une instruction `awk` est composée d'un ou des deux éléments :

motif `{`*action*`}`

Chaque ligne du fichier traité est comparée avec les *motifs*, un par un, l'*action* correspondante étant exécutée la cas échéant.

Si le *motif* est absent, l'*action* est effectuée pour chaque ligne du fichier.

Si l'*action* est omise, chaque ligne correspondant au *motif* est imprimée sur la sortie standard

23.1 Les variables prédéfinies

`awk` traite le fichier d'entrée par enregistrement. Le séparateur d'enregistrement est contenu dans la variable de type caractère `RS`. Chaque enregistrement est composé de champs séparés par le contenu de la variable `FS`, également de type caractère.

Pour le fichier de sortie, les correspondants de ces deux variables sont `ORS` et `OFS`

La variable `NR` contient le numéro de l'enregistrement en cours de traitement.

Lorsque `awk` traite un enregistrement, il assigne le contenu du $n^{\text{ième}}$ champ à la variable `$`n, le nombre de champ à `NF` et le contenu total de l'enregistrement à `$0`.

23.2 Les motifs

23.2.1 Motifs particuliers

Deux motifs permettent d'initialiser des paramètres du traitement ou d'effectuer des calculs statistiques finaux :

`BEGIN` permet de spécifier des actions à effectuer avant de commencer à lire le fichier d'entrée.

Exemple :

```
BEGIN {FS=" :"}
```

`END` permet de spécifier des actions à effectuer après que le fichier d'entrée ait été traité.

Exemple :

```
END {print NR}
```

23.2.2 Expression régulière

Une expression régulière est entourée de slashes :

Exemple :

```
/^[a-z]/
```

permet d'imprimer toutes les lignes commençant par une minuscule.

23.2.3 Correspondance de chaîne

On peut spécifier le champ contenant une chaîne, ou ne la contenant pas avec `~` et `!~`

Exemple :

```
$4 !~ /[0-9][0-9][0-9]/
```

spécifie toutes les lignes dont le quatrième champ ne contient pas 3 chiffres.

23.2.4 Relations arithmétiques entre champs

On peut exprimer des relations avec les opérateurs `<` , `<=` , `==` , `!=` , `>=` et `>`

Exemple :

```
$1 == 1990
$2 >= $4 + 100 {print $2}
```

imprime les lignes dont le premier champ vaut 1990, puis le second champ des lignes où la valeur de celui-ci dépasse d'au moins 100 la valeur du quatrième.

23.2.5 Opérations booléennes entre expressions

Certaines expressions peuvent être combinées avec les opérateurs `&&` (et), `||` (ou) et `!~(non)`

Exemple :

```
($1 > "r" && $1 < "u") || $2 == "Mar"
```

spécifie les lignes dont le premier champ commence par « s » ou « t » ou le second par « Mar ».

23.2.6 Domaine de valeur

Un domaine de valeur peut être exprimé en donnant les limites séparées par une virgule :

Exemple :

```
/10 janvier/ , /19 janvier/
NR == 5 , NR == 17
```

indiquent de prendre en considération toutes les lignes situées entre celles contenant la chaîne « 10 janvier » et celles contenant la chaîne « 19 janvier », ainsi que les lignes 5 à 17

23.3 Les actions

Une action est entourée d'accolades. Elle peut contenir plusieurs instructions, et celles-ci peuvent être données sur des lignes consécutives, ou séparées par un point-virgule « ; » .

Une instruction peut être remplacée dans certains cas par plusieurs instructions entourées d'accolades. De tels groupes sont notamment utilisés avec les structures de contrôle (voir plus loin).

23.3.1 Impressions

On peut imprimer une ligne, un champ, une chaîne, une valeur ou le contenu d'une variable sur la sortie standard avec la commande : `print`.

Exemple :

```
{print NR "> " $0}
```

imprime la ligne pointée par son numéro.

On peut formatter une ligne, un champ ou le contenu d'une variable et l'envoyer sur la sortie standard ou dans une chaîne avec les fonctions : `printf` et `sprintf`.

Exemple :

```
{printf("%d> %s",NR,$0)}
{head = sprintf("%d> ",NR)  ; print head $0}
```

sont deux autres manières d'imprimer la ligne pointée par son numéro.

23.3.2 Exécution de fonctions

On peut utiliser dans la partie active de awk les fonctions : `sqrt(arg)` , `exp(arg)` , `log(arg)` , `int(arg)` , `abs(arg)` , `length[(str)]` , `substr(str,n1,n2)` et `index(s1,s2)`

Exemple :

```
{print abs($5)}
```

imprime la valeur absolue du cinquième champ.

23.3.3 Assignation de variables

On peut assigner des valeurs à des variables internes, les raccourcis `+=` , `-=` , `/=` et `*=` peuvent être utilisés.

Exemple :

```
{y = substr($6,1,length($6) - 2)}
{sum += $4}
```

met dans la variable `y` la chaîne contenue dans le champ 6, sauf les deux derniers caractères, et somme le champ 4 à des fins statistiques dans la variable `sum`.

23.3.4 Assignation de champs

On peut assigner une valeur aux champs de sortie. Le champ modifié prend la place du champ correspondant dans l'enregistrement, sans modifier le fichier d'entrée.

Exemple :

```
{$3 = $1 + abs($2)  ; print $0}
```

imprime les lignes, avec le troisième champ égal à la somme du champ 1 et de la valeur absolue du champ 2.

23.3.5 Structures de contrôle

On peut modifier le déroulement d'un programme en utilisant les instructions de contrôle `if-else` , `while` et `for`

Exemple :

```
{if ($2 >= 0)
  print $2
else
  print -$2
}
```

imprime la valeur absolue du deuxième champ.

Si les lignes d'un fichier présentent en colonne 1 une somme, en colonne 2 un taux d'intérêt annuel et en colonne 3 un nombre d'années, le programme :

```
{i=1
 while(i <= $3) {
  print $1 * (1 + $2)^i)
  i += 1
 }
}
```

imprime les intérêts composés à la fin de chaque année, jusqu'à la date d'échéance.

```
{sum=0
 for(i = 1  ; i <= NF  ;i = i+1) sum = sum + $i
  print sum
}
```

imprime la somme des champs de la ligne.

23.3.6 Exemple complet

Imprimons le nom réel de tous les utilisateurs du groupe 4700 où l'on trouve tous les chercheurs de l'Observatoire, sans utiliser le champ de groupe dans le fichier `passwd` :

```
pegase:~ 92) niscat passwd | awk 'BEGIN {FS=" :" ;ORS=", "}\
$3~/47[0-9][0-9]/ {print $5}'
BARBLAN Fabio, SIMOND Gilles, MEYNET Georges, FLEURY Miche
```

l, ISCHI Emile, GOLAY Marcel, FRIEDLI Daniel, LECHAIRE Dan
iel, WALTER Roland, MAEDER Andre, DEPARTEMENT Mecanique, B
URNET Michel, MAIRE Charles, MEGEVAND Denis, DUQUENNOY Ant
oine, CRAUSAZ Michel, MERMILLIOD Monique, HAUCK Bernard, D
E MEDEIROS Jose Renan, SCHALLER Gerard, QUELOZ Didier, UDR
Y Stephane, ESIS Astronomy dB, NITSCHELM Christian, NORTH
Pierre, CRAMER Noel, BETRIX Franck, LANZ Thierry, BRATSCHI
Pierre, RICHARD Christian, GENEVAY Olivier, COURVOISIER T
hierry, MIDAS Albert, WILHELM Barbara, REICHEN Michael, NI
COLET Bernard, LAMPENS Patricia, BLECHA Andre, MAYOR Miche
l, BABEL Jacques, SCHAERER Daniel, DUBOSSON rene, PERNIER
Bernard, DUBATH Pierre, BUHLER PAUL, CHMIELEWSKI Yves, GOY
Gerald, PFENNIGER Daniel, PALTANI Stephane, GEORGE Michel
, BARTHOLDI Paul, ORR Astrid, CAMERA, DUQUENNOY Antoine, P
FENNIGER Daniel, FRIEDLI Daniel, BERTHET Stephane, GRENON
Michel, BURKI Gilbert, FUX Roger, COURVOISIER Thierry, MER
MILLIOD Jean-Claude, ELODIE Elodie, PROJET telescope, RUSS
INIELLO Giovanni, MARTINET Louis, HUGUENIN Daniel, WEBER L
uc,

Chapitre 24

make

Outil facilitant le développement et la maintenance de programmes.

`make [-dinpst][-f` *fichier*`][`*ARG*=*chaîne*...`][`*cible*...`]`

`make` permet au programmeur de développer des programmes complexes, sans avoir à se préoccuper de tout recompiler à chaque modification, ou de se souvenir de toutes les dépendances entre routines à chaque compilation. `make` permet également de développer des programmes simples sans avoir besoin de taper de longues lignes de commandes pour la compilation, l'édition des liens ou d'autres outils de programmation.

Options :

`-d` imprime les décisions et des informations sur la manière dont elles ont été prises.

`-f` utilise le *fichier* comme *makefile*.

`-i` ignore les erreurs retournées par les commandes, et continue.

`-n` imprime les commandes sans les exécuter.

`-p` imprime les macros et les description des cibles.

`-s` n'imprime pas les commandes avant de les exécuter.

`-t` met-à-jour les dates des cibles, sans les reconstruire.

Les *ARG* permettent de passer des macros-définitions au *makefile*.

24.1 Introduction

Lorsque l'on crée un programme, il faut d'abord écrire les fichiers sources, compiler ces sources, puis combiner les codes objets produits avec les librairies contenant les routines appelées (édition des liens) pour obtenir un exécutable.

Ce processus, s'il n'est pas forcément fastidieux pour de petits programmes, peut le devenir si plusieurs librairies de routines doivent être liées au programme, et qu'on est en phase de développement, où l'on modifie et recompile fréquemment son code.

Lorsque le code est important, réparti dans plusieurs modules, et donnant éventuellement même lieu à plusieurs exécutables, il est pratique d'avoir un outil qui permette de ne recompiler que les modules modifiés et/ou ceux qui en dépendent. Un gain de temps et de ressources de calcul important peut être le résultat de l'utilisation de `make`.

24.2 Le makefile

`make` lit un fichier (*makefile*) dont le nom est `makefile`, ou `Makefile` dans le répertoire courant, à moins que le nom ne soit spécifié sur la ligne de commande.

Un *makefile* peut construire plusieurs cibles, qui peuvent être des programmes, mais aussi d'autres choses que l'on demande au système de gérer. Une cible habituelle est un programme exécutable.

Dans le *makefile* sont décrites les dépendances entre modules du programme, ainsi que les règles de construction des cibles.

`make` examine en fonction de ces dépendances les dates des fichiers correspondants, et décide si une reconstruction est nécessaire. Si c'est le cas, les règles fournies ou par défaut sont appliquées.

Si par exemple un fichier source est plus récent que le code objet qu'il produit (on dit que le code objet est la cible, et le source sa dépendance), `make` comprend que le source a été modifié, et le recompile. L'exécutable devient alors plus ancien que le code objet, et `make` fera appel à l'éditeur de liens `ld` pour le recréer.

24.3 La structure du makefile

Un *makefile* est composé de trois types d'instructions : les macros-définitions, les dépendances et les règles.

24.3.1 Les dépendances

Chaque cible est donnée en début de ligne, suivie du signe « : », et des noms de fichiers à partir desquels elle est construite, qu'on appelle les dépendances.

Exemple :

```
test :      test.o screen.o keyboard.o
```

Dans un module C, on peut inclure des définitions contenues dans un fichier d'entête (`.h`). On indique pour les programmes C, en fin de *makefile*, la liste des entêtes, comme des dépendances des codes objets. `make` utilisera ses règles par défaut pour compiler le source avec les entêtes correspondantes si besoin est.

Pour un autre langage, par exemple le Fortran, on doit indiquer également le fichier source, pour que `make` puisse déterminer la règle de compilation (voir règles par défaut, ci-dessous).

Exemple :

```
test.o :      test.h
screen.o keyboard.o :      vt100.h sun.h atari.h
```

24.3.2 Les règles

Pour construire chaque cible, on exécute une ou une série de commandes, qui sont spécifiées après les dépendances. Les lignes de règles doivent commencer par un `<TAB>`, puis présenter la commande telle qu'on l'écrirait dans le shell.

Exemple :

```
cc -o test test.o screen.o keyboard.o -ldevices
```

`make` utilise un certain nombre de règles par défaut. `make` utilise le suffixe des noms de la cible et de ses dépendances pour déterminer laquelle de ces règles est appropriée.

24.3.3 Makefile minimum de l'exemple

```
test :      test.o screen.o keyboard.o
      cc -o test test.o screen.o keyboard.o -ldevices

test.o :      test.h
screen.o keyboard.o :      vt100.h sun.h atari.h
```

24.3.4 Les macros-définition

On peut inclure des noms permettant de remplacer une chaîne, et ainsi d'éviter de retaper plusieurs fois toute cette chaîne. De plus, l'habitude de donner des noms en majuscules à ces macros-définitions permet de les faire ressortir du texte du *makefile*, et d'en repérer rapidement certaines parties.

Le nom de la *macro* doit être suivi du signe « = », puis de la chaîne de remplacement.

Exemple :

```
OBJ = test.o screen.o keyboard.o
```

On les utilise ensuite dans le reste du *makefile* en les mettant entre parenthèses, le tout précedé du signe « $ ».

Exemple :

```
test : $(OBJ)
      cc -o test $(OBJ) -ldevices
```

Un certain nombre de *macros* de base sont aussi définies, que l'on peut utiliser dans le *makefile*, dont notamment :

```
CC=cc
CFLAGS=
CPPFLAGS=
LINT=lint
LINTFLAGS=
COMPILE.c=$(CC) $(CFLAGS) $(CPPFLAGS)  -c
LINK.c=$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)
FC=f77
FFLAGS=
COMPILE.f=$(FC) $(FFLAGS) -c
LINK.f=$(FC) $(FFLAGS) $(LDFLAGS)
LD=ld
```

```
LDFLAGS=
LDLIBS=
MAKE=make
RM=rm -f
AR=ar
ARFLAGS=rv
```

24.3.5 Les commentaires

On peut également introduire des commentaires dans un *makefile*, sur des lignes commencant par le signe « # » .

Exemple :

```
# Ce makefile produit le programme test
# Il utilise la librairie libdevices.a
# situee dans /usr/local/lib.
```

24.3.6 Makefile final de l'exemple

```
# Ce makefile produit le programme test
# Il utilise la librairie libdevices.a
# situee dans /usr/local/lib.

CFLAGS=-O
OBJ = test.o screen.o keyboard.o

test : $(OBJ)
      $(LINK.c) -o test $(OBJ) -ldevices
test.o :      test.h
screen.o keyboard.o :      vt100.h sun.h atari.h
```

24.3.7 Remarques

– Les macros-définitions permettent de gérer en un seul endroit les paramètres éventuellement modifiables du *makefile*. Ceci devient important lorsque l'on écrit un *makefile* important contenant plusieurs cible de même type.

– `make` tentera par défaut de construire la première cible du *makefile*. Pour construire une autre cible, il faut donner la commande :

  `make` *cible*

  Exemple :

  ```
  pegase:~ 93) make test.o
  ```

  ne construira que le code objet du module principal de l'exemple.

– On peut construire un petit *makefile* générique qui utilise le passage des macros-définitions à partir de la ligne de commande. Le fichier `myformake` suivant permet de compiler un source fortran de manière facile :

```
# Makefile generique pour compiler en fortran.
# Utilise la macro NAME fournie par la commande
```

```
# et les librairies de supermongo et libm.a

LDLIBS = -lplotsub -ldevices -lutils -lmiss -lm
# NAME = (parametre de la ligne de commande)

$(NAME) : $(NAME).f
	$(LINK.f) -o $(NAME) $(NAME).f $(LDLIBS)
```

Si on lance la commande :

```
pegase:~ 94) make -f myformake NAME=mytest
```

Le fichier `mytest.f` sera compilé, et donnera un exécutable `mytest`. Si, de plus l'on définit un alias `for` comme :

```
pegase:~ 95) alias for "make -f myformake NAME=\!*"
```

on pourra lancer la construction de (`mynewtest.f`) avec la ligne

```
pegase:~ 96) for mynewtest
```

Chapitre A
Corrections des exercices

Exercices du chapitre 1 :

1.1 Sous **SunOS 4.x**avec les NIS, utilisez la commande :

`ypmatch {\em machine} hosts`.

La commande `hostname` vous indiquera le nom de la *machine*. Habituez-vous à faire directement

`ypmatch ‘hostname‘ hosts`.

Sous **SunOS 5.x**, avec les NIS+, utilisez la commande

`nismatch ‘hostname‘ hosts.org_dir`.

1.2 Le numéro de groupe est le quatrième champ des lignes du fichier `passwd`. Donc il suffit d'utiliser la commande grep (voir chapitre page 95), la commande `ypmatch` (NIS) ou `nismatch` (NIS+) avec son nom d'utilisateur (`‘whoami‘`) sur le fichier ou la table `passwd(.org_dir)`. Par exemple

```
nismatch ‘‘whoami‘ passwd.org_dir|awk -F: ’{print $4}’
```

Exercices du chapitre 3 :

3.1 `cd`, `cd ~`, `cd $HOME`

3.2 L'entrée `..` du repertoire de base `/` pointe sur le descripteur du fichier `/` lui-même.

Exercices du chapitre 4 :

4.1 Imprime `hello, world` à l'écran.

4.2 Imprime `hello, world` dans le fichier `test`.

4.3 Liste le contenu du répertoire `/usr/lib` à l'écran.

4.4 Liste le contenu du répertoire `/usr/lib` par écrans successifs.

4.5 Liste le contenu du répertoire `/usr/lib` dans le fichier `more`.

4.6 Liste le contenu du fichier `test` par écrans successifs.

4.7 Rend exécutable le fichier `more`.

4.8 Exécute la première commande `more` du chemin de recherche des exécutables. Si le répertoire courant est examiné avant les repertoires du système, cela pourrait conduire à tenter d'éxecuter le fichier créé en 4.5, ce qui donnera plus de messages d'erreurs qu'autre chose. **Selon le contenu d'un tel fichier `more`, cela peut conduire à une catastrophe.**

Exercices du chapitre 6 :

6.1 `whoami` donne le nom de l'utilisateur, alors que `who am i` détaille la session en précisant où et quand l'utilisateur s'est connecté, et quel terminal lui est attribué.

6.2 Découvrez-le en utilisant la commande `du`.

Exercices du chapitre 7 :

7.1 Lorsque le repertoire courant (`.`) ou parent (`..`) a été détruit par un autre processus. Si le répertoire est sur un disque local, on se retrouve dans le parent de son répertoire de base (`~/..`), si on est sur un disque monté par NFS, on obtient le message d'erreur `..: Stale NFS file handle.`

7.2 `$ mkdir cours ; cd cours`

Exercices du chapitre 8 :

8.1 `$ cp -p ~/.login test1`

8.2 `$ tr "[A-Z]" "[a-z]" < test1 > test2 ; rm test1`

8.3 `$ mkdir cours-unix ; mv * cours-unix ; cp ~/.login cours-unix`

8.4 `$ rm -r cours-unix ; pwd ; cd`

Exercices du chapitre 9 :

9.1 `$ ls -ld /usr/local` donne une ligne concernant le répertoire `/usr/local`. Le nombre de liens sur un répertoire est égal au nombre de pointeur sur son descripteur. Outre l'entrée `local` dans le repertoire `/usr` et l'entrée `.` de `/usr/local`, il reste les entrées `..` des sous-répertoires de `/usr/local`, dont le nombre vaut $n-2$, n étant le chiffre indiqué sur la ligne du `ls -ld`.

9.2 Mettez la ligne suivante dans le fichier magic, les champs sont séparés par des <TAB> :

```
1       string          documentstyle   LaTeX 2.09 document
```

Exercices du chapitre 10 :

10.1 `$ touch essai ; ls -l essai`

10.2 `500` ou `u=rx,go=`

Exercices du chapitre 11 :

11.1 `$ cat .alias*`

11.2 `$ man csh`, puis pendant l'affichage, recherchez la partie concernée avec `/history`, et d'eventuels `/` supplémentaires.

Exercices du chapitre 12 :

12.1 Notez les différences de format de sortie, ainsi que le comportement face à des fichiers binaires.

Exercices du chapitre 13 :

13.1 `$ niscat passwd.org_dir|wc -l` ou
`$ ypcat passwd | wc -l` selon que les NIS ou les NIS+ sont installées.

Exercices du chapitre 18 :

18.1 `$ ps -eu ‘whoami‘ | wc -l` sous **SunOS 5.x**,
`$ ps -ux  | wc -l` sous **SunOS 4.x**.

18.2 En lançant la commande `jobs`.

Index